Unix System Programming with Standard ML

Unix System Programming with Standard ML

Version: 0.1, Mar 2002

Table of Contents

List of Tables

List of Figures

Preface

This book is a study in system programming using the Standard ML language. (In the rest of the book this language will be referred to as SML). Standard ML is one of several languages in the ML family, the other main one being OCaml[*OCaml*], but this book concentrates on Standard ML and in particular the Standard ML of New Jersey (SML/NJ) implementation. I will revisit some related languages in the concluding chapter.

SML belongs to the class of functional programming languages. Functional languages are very high-level languages that are often thought of as academic curiosities, not up to the job of real-world programming. This book aims to show that functional languages, in particular SML, can be used for real-world programs.

The definition of system programming is a bit fuzzy. This book will only cover programming in the Unix operating system. In Unix, by system programming, I mean being able to write infrastructure programs such as daemons and utilities that interact with other programs, not necessarily directly with the user.

The Audience

I've seen many postings on Usenet news groups and mailing lists by programmers interested in using functional languages who have been unable to get very far due to the paucity of documentation and training material. What material on functional programming that is available tends to be either rather academic or only at an introductory level with small toy examples. This book aims at improving this situation by working through a project more typical of real-world programs. In addition there will be smaller examples in tutorial chapters.

The major project in this book is a web server. It is similar in functionality to the early CERN server. It supports CGI programming but only does the

original HTTP 1.0 protocol.

You, dear reader, should be a programmer familiar with Unix programming using conventional languages such as C, C++ or Java. You have probably been introduced to functional programming at college and retained an interest in the subject but not had the opportunity to take it further. Microsoft Windows programmers with some familiarity with the Posix standards will benefit from this book too.

You should already have some familiarity with SML or other languages in the ML family, or enough familiarity with functional languages that you can pick up the language from a tutorial. This book is not a course on SML. There are a number of books available that teach programming in SML. There are also some tutorials available on the web. See Appendix A for more information on what's available.

The Environment

There are several implementations of SML available. The flagship implementation is the New Jersey implementation from Bell Laboratories, now a division of Lucent Technologies. This implementation is called SML/NJ. It is available for a variety of machine architectures running some flavour of Unix. It is also available for Microsoft Windows. All of the examples in this book have been tested on Linux using version 110.0.7. You can get it from [*SML*].

SML/NJ comes with a concurrent programming library known as Concurrent ML or CML. The major project in this book will be written as a concurrent program using CML.

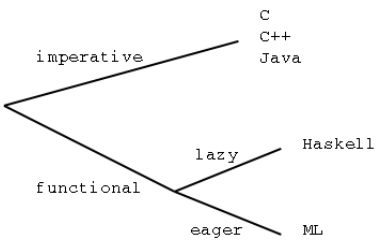
I. Programming with Standard ML

Chapter 1. Introduction

What is Functional Programming?

This section gives a brief overview of functional programming to set the scene. Figure 1-1 shows a rather simplistic classification of some contemporary programming languages.

Figure 1-1. A Classification of Some Languages



The conventional languages are based on the imperative programming paradigm (which I'll abbreviate to IMP). This means that the program progresses by applying commands to a store containing the program variables. The commands cause updates to the contents of the store. As the program executes the store goes through a sequence of states.

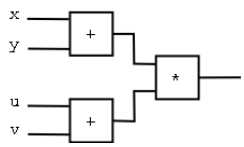
In functional programming (which I'll abbreviate to FP) there is no store that is updated. The program progresses by computing new values from existing values. There is nothing in pure FP that requires the existence of a distinct store let alone an updatable one.

FP is further divided into two main camps, eager and lazy. Eager languages always evaluate an expression at the earliest opportunity. All inputs to functions are values. The ML language family is eager.

Lazy languages delay the evaluation of an expression until its value is actually needed. The aim is to avoid computing unneeded values. This means that inputs to functions may be partially evaluated expressions. A very useful consequence of lazy evaluation is that it becomes possible to deal with data structures of infinite size, as long as you don't need to evaluate all of the data structure. The flagship lazy functional language is Haskell[*Haskell*] with Clean[*Clean*] an important rival.

It might seem that there is not much difference between IMP and FP since they both involve computing new values from existing values. The difference is best illustrated graphically. Figure 1-2 shows a simple expression and a graphical representation of the values and operators (functions).

Figure 1-2. The expression `(x + y) * (u + v)`



This expression illustrates several essential features of FP.

- Values flow between operators. The values are immutable by nature. A value such as 3 is always 3 and never anything else.
- Operators always compute the same result given the same inputs (at least within the same environment). If you compute 1+2 at one point in time and do it again later you can expect to get the same result.
- The only effect of an operator is to produce an output value. Nothing else is disturbed in the system.

Functions that always compute the same results for the same inputs are called *pure* or *referentially transparent*. Functions that have no other effect than to produce their output value are called *side-effect free*. These properties are closely connected. If a function's behaviour depends on some input other

than its manifest inputs then some other function could change this other input as a side-effect. In this case the behaviour of the function can change at times and in ways that can be very difficult to predict and hard to debug.

FP says that these features of immutable values and pure, side-effect free functions are so valuable that they must be preserved throughout the design of the language.

IMP programs abandon these features as soon as the result of the expression is assigned to a variable. A variable in IMP is a name bound to a box into which different values can be placed at different times. This variable can be updated by a side-effect of some function `f`. The behaviour of a function `g` in another part of the program can depend on the value of this variable. These two functions are communicating a value using the variable as a channel. I call this a *sneak path*. The sneak path is not obvious from the source code. There is nothing to see at the place where `f` is called to indicate that it is making this communication. IMP programs are full of sneak paths. Part of the challenge of figuring out an IMP program is tracing all the possible sneak paths. IMP programmers are warned not to use lots of global variables in their programs as it makes the scope of the sneak paths enormous. But that only barely contains the problem.

A further problem with communication via a variable is that its correctness depends on the order of operations. The call to `f` must be made before the call to `g` or else it fails. You've probably had many bugs in your programs like this where you've got something happening in the wrong order. I sure have.

In general, since variables go through sequences of states, getting computation steps in the right order is essential to IMP. Controlling the order of operation is a large part of what makes IMP difficult. You find that you need to use a debugger to observe the order of events in the program and the changes in variables just to figure out what is going on.

Pure FP eliminates these problems.

- The meaning of the word *variable* in FP is different. It is a name bound to a value, not to a box holding a value. So the value associated with the

variable is steady.

- All inputs to a function are made manifest in its declaration as argument variables. For convenience a function may depend on the value of a variable in a surrounding scope but this value must by definition be steady during the lifetime of the function so the behaviour of the function cannot vary over its lifetime.
- The communication between functions is more directly visible in the source code.
- There is no order of evaluation to worry about. Referring back to Figure 1-2, you see that it doesn't matter in what order the addition operators are performed. The only ordering that is needed is that a function does not compute until all its input values are available. This order can be determined automatically by the language system so you don't need to worry about it at all.

These features make it a lot easier to reason rigourously about FP code. A simple demonstration of the difficulty with IMP is the mathematical rule `x + x = 2x`. You would think that any logical system that violates even as simple a rule as this is going to be hard to work with. So consider if this C expression can be true: `getchar() + getchar() == 2*getchar()`!

Unfortunately, while pure FP is very nice in theory, it has its problems in practice. Some of these are described in the next sections.

Pure FP and I/O

The major problem with pure FP is input/output (I/O). Except in the simplest cases where a program can be cast as a function transforming one file into another, I/O is a side-effect. For example, an FP program could have sneak-paths by writing to a file and reading it back again later. The outer world, the operating system, is a vast mess of state that the program has to interact with.

There are a variety of techniques that have been developed in recent years to find a way to get pure FP and I/O to blend together. The most advanced can be found in the Haskell and Clean languages. I won't go into the details except to mention the idea of lazy streams.

A lazy stream is an infinite list of values that is computed lazily. The stream of keystrokes that a user presses on the keyboard can be represented as an infinite (or arbitrarily long) list of all of the keystrokes that the user is ever going to press. You get the next keystroke by taking the head element of the stream and saving the remainder. Since the head is only computed lazily, on demand, this will result in a call to the operating system to get the next keystroke.

What's special about the lazy stream approach is that you can treat the entire stream as a value and pass it around in your program. You can pass it as an argument to a function or save the stream in a data structure. You can write functions that operate on the stream as a whole. You can write a word scanner as a function, `toWords`, that transforms a stream of characters into a stream of words. A program to obtain the next 100 words from the standard input might look something like

```
apply show (take 100 (toWords stdIn))
```

where `stdIn` is the input stream and `take n` is a function that returns the first `n` elements in a list. The `show` function is applied to each word to print it out. The program is just the simple composition of the functions. Lazy evaluation ensures that this program is equivalent to the set of nested loops that you would write for this in an IMP program. This is programming at a much higher level than you typically get with IMP languages.

But we aren't using Haskell or Clean. The approach of SML to I/O is to revert to impure behaviour. You can use a `read` function and it will return different values on each call. You can use a `print` function and it will have the side-effect of writing to a file. You can even have *imperative variables* with assignment statements. These are called reference types. They provide you with a box into which you can store different values at different times. So this means that the sequencing of statements raises its ugly head in SML.

With SML you can use a mixture of FP and IMP but naturally the IMP should be kept to a minimum to maximise the advantages of FP. When we get into concurrent programming later on we'll look at ways of blending the two. Communication between threads or coroutines can be used to emulate lazy streams.

Pure FP and Plumbing

One of the awkward problems with pure FP is that all input values to a function must be passed in as arguments at some point. This creates the plumbing problem. If you want to pass a value to some point in the program it must be plumbed through all of the functions in between which can lead to a lot of clutter. This problem can be reduced a bit by taking advantage of nested functions and scoping. In the following code

```
fun f a b c =
let
    fun g x count =
    (
        ... g (x-1) (count+1) ...
    )
in
    g b 1
end
```

the values `a` and `c` are available to the function `g` without having to be passed as arguments. But even doing this it can be sometimes be quite awkward plumbing everything through. In practice, the careful use of some global imperative variables can improve a program's readability a lot, as long as we don't have sneak paths as an essential feature of an algorithm.

A second part to the plumbing problem is the chaining of values from one function to the next. You often have code looking something like

```
fun f a =
let
    ...
    val (x, s1) = g a []
```

```
    val (y, s2) = g b s1
    val (z, s3) = g c s2
    ...
in
    ...
end
```

Here the functions are computing values `x`, `y` and `z` while passing some state values between the calls. There is no easy solution to this in SML. The Haskell language has a design pattern called a *monad* which neatly simplifies this plumbing. In SML you would have to roll your own monads which is messy enough that you might as well just grin and bear it.

Chapter 2. Hello World

In this chapter I will show the basics of running simple SML programs. I will start with the classic *hello world* program to show how to build a complete runnable program. Then I will move on to more elaborate programs following the classic development with programs like *echo* and *word count*.

During this development I will stop to examine some of the programming idioms that are peculiar to functional programming and that often give an imperative programmer difficulties. I will pay particular attention to loops, using recursion, which is one of the biggest differences between imperative and functional programming.

By the time you get here you should have studied one of the texts or tutorials on SML cited in Appendix A.

All programming examples are available as complete source files that you can run. I only cite pieces of each program in the text. You should also read the complete programs.

Assembling the Hello World Program

The least you have to do to make SML say "hello world" is to use the *top level*. This is a classic Read-Eval-Print loop where you type in an SML declaration and it is immediately evaluated. For example

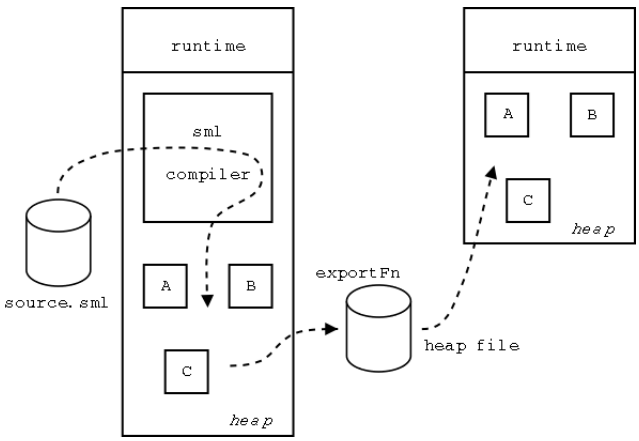
```
> sml
Standard ML of New Jersey, Version 110.0.7
- print "hello world\n";
hello world
val it = () : unit
- ^D
```

You type in the `print` expression at the prompt and terminate it with a semicolon. It gets evaluated which has the side-effect of printing the string. Then the compiler shows the return value which is `()` of type `unit` and is

assigned to the special variable `it`. The type `unit` plays the same role as `void` in C. Use a Control-D to exit from the compiler.

But this doesn't get you a program that you can run as a command. For this you need to do some more work. The SML/NJ system can save a program in an (almost) ready-to-run form called a heap file. The steps it goes through are shown in Figure 2-1

Figure 2-1. Compiling and Running a Program



When you run the `sml` command you start off with the runtime, which is a C executable, and a heap containing the compiler. The compiler compiles your source file to one or more modules, here A, B and C, which are left in the heap. Then you arrange for the contents of the heap to be dumped into a file using the built-in `exportFn` function. Just before the heap is dumped a garbage collection is performed which removes all objects in the heap that are not reachable from your program. This will get rid of the compiler.

Later you can run your program by starting another copy of the runtime and loading your program into its heap.

You can find out where the runtime executable is on your computer by looking for where the `sml` command is kept. In my installation I have these

files.

```
> which sml
/usr/local/bin/sml

> ls -l /usr/local/bin/sml
... /usr/local/bin/sml -> /src/smlnj/current/bin/sml

> cd /src/smlnj/current/bin
> ls -a
./
../
.arch-n-opsys*
.heap/
.run/
.run-sml*
ml-burg -> .run-sml*
ml-lex -> .run-sml*
ml-yacc -> .run-sml*
sml -> .run-sml*
sml-cm -> .run-sml*
```

The `sml` command leads back to the `.run-sml` shell script. This script runs the executable in the `.run` subdirectory with a heap file that contains the compiler, found in the `.heap` subdirectory.

To build your own program you need to duplicate this arrangement. Here is a basic *hello world* program.

```
structure Main=
struct

    fun main(arg0, argv) =
    (
        print "hello world\n";
        OS.Process.success
    )

    val _ = SMLofNJ.exportFn("hw", main)
end
```

All compiled programs are divided into modules called *structures*. Here I've called it `Main` but the name doesn't matter. After the structure is compiled each of its declarations will be evaluated. Evaluating the function `main`

doesn't do anything but say that this is a function. But when the `val` declaration is evaluated the `exportFn` function (in the built-in structure `SMLofNJ`) will be called. This will write the heap into a file named `hw` with a suffix that depends on the kind of operating system and architecture you are using. For Linux on ix86 the file name will be `hw.x86-linux`.

The second argument to `exportFn` names the function that will be called when the heap file is read back in. This function must return a success or fail code which becomes the exit code (0 or 1) of the program. These codes are defined in the built-in `OS.Process` structure.

The next step is to compile this program. The SML/NJ system comes with a built-in compilation manager that does a job similar to the Unix `make` command. First you need a CM file that describes what you are going to compile. Call it `hw.cm`. The least it needs to contain is

```
group is
    hw.sml
```

Then compile the program as follows[1]

```
> CM_ROOT=hw.cm sml
Standard ML of New Jersey, Version 110.0.7, September 28, 2000
- CM.make();
[starting dependency analysis]
[scanning hw.cm]
[checking CM/x86-unix/hw.cm.stable ... not usable]
[parsing hw.sml]
[Creating directory CM/DEPEND ...]
[dependency analysis completed]
[compiling hw.sml -> CM/x86-unix/hw.sml.bin]
[Creating directory CM/x86-unix ...]
[wrote CM/x86-unix/hw.sml.bin]
GC #1.1.1.1.1.10:   (10 ms)
write 1,0: 1356 bytes [0x40cd0000..0x40cd054c) @ 0x1000
........ stuff deleted
write 5,0: 28 big objects (271 pages) @ 0x15410
```

The most convenient way to pass in the name of the CM file is through the `CM_ROOT` environment variable. (If you don't set `CM_ROOT` then a default of `sources.cm` is used.) At the prompt type `CM.make();`. This runs the

compilation manager. Don't forget the semicolon. You will be prompted until you enter it.

The messages you get show the compilation manager figuring out that it needs to recompile the source file. It then caches the compiled form in the `CM/x86-unix/hw.sml.bin` file. Then the export step writes lots of stuff to the heap file.

Now that you have the heap file you need a shell script to run it. Here is a generic script.

```
heap=‘basename $0‘
install=.
smlbin=/src/smlnj/current/bin

exec $smlbin/.run-sml @SMLload=$install/${heap}.x86-linux "$@"
```

This script starts the runtime and specifies the heap file to load. This is taken from the name of the script so that the same script can be used for different programs by adding links or just copying it. The `install` variable allows you to move the heap file to some installation directory. Any command line arguments will be passed through to the SML main function.

Now you can run it

```
> hw
hello world
```

Don't be too worried about the large size of the heap file for such a small program. Some people argue that there is something wrong with a language if a program as small as this doesn't produce a correspondingly small executable and cry *Bloat!*. But few people write *hello world* programs. The overhead in the heap file becomes much more modest in proportion when you develop programs of a serious size.

That's a fair bit of work to get a single program going but you only have to do it once and copy it as a template for future programs.

The *echo* Program

The *echo* program will just echo its arguments. It's a simple extension to *hello world* that demonstrates using the command line arguments. Here is the program.

```
structure Main=
struct

    fun main(arg0, argv) =
    (
        case argv of
          [] => ()

        | (first::rest) =>
        (
            print first;
            app (fn arg => (print " "; print arg)) rest;
            print "\n"
        );

        OS.Process.success
    )

    val _ = SMLofNJ.exportFn("echo", main)
end
```

The name of the program is supplied to the main function as the first argument, `arg0` of type `string`. This is the same as `argv[0]` in a C main function. The remaining command line arguments are supplied as a list of strings in `argv`.

The `case` expression reads as follows. If the argument list is empty then print nothing. Otherwise if it is non-empty then print the first one and then print the rest of them preceded by a single blank. The `app` applies the function to each of the strings in the list `rest`. This function prints the blank and then the argument.

Be careful that you only put semicolons *between* imperative statements. If you were to put a semicolon after the new-line print you would get this error message from the compiler.

```
echo.sml:14.2 Error: syntax error: inserting  EQUALOP
```

This is because the parser in the compiler attempts to correct syntax errors in order to continue compiling for as long as possible. Sometimes this works. In this case it decides that it needs another expression after the semicolon and chooses an equals operator to be the expression. The error message points to the line after where the surplus semicolon is, here line 14, character 2. See Appendix B for more information on the compiler's error messages.

For this program I've used a more elaborate run script since there is a bug in the standard one in that it does not pass empty command line arguments through to the SML program. For the *echo* program to be exactly right we need to see the empty arguments too. This script copies in more of the SML/NJ `.run-sml` script.

```
install=.
smlbin=/src/smlnj/current/bin

arg0=$0
cmd=‘basename $0‘

ARCH_N_OPSYS=‘$smlbin/.arch-n-opsys‘
if [ "$?" != "0" ]; then
    echo "$cmd: unable to determine architecture/operating system"
    exit 1
fi
eval $ARCH_N_OPSYS
suffix=$ARCH-$OPSYS

exec $smlbin/.run/run.$ARCH-$OPSYS @SMLcommand=$arg0 \
        @SMLload=$install/${cmd}.$HEAP_SUFFIX "$@"
```

Loops and Recursion

The Basics

The typical imperative loop idiom is demonstrated by this C function for finding the sum of the nodes in a list.

```
typedef struct node {
    int          value;
    struct node* next;
} Node;

int
sumlist(const Node* the_list)
{
    int     sum = 0;
    Node*   l;

    for(l = the_list; l != 0; l = l->next)
    {
        sum = sum + l->value;
    }

    return sum;
}
```

This shows two major features of imperative loops. There is a loop variable, here `l`, that passes through a sequence of values, one for each iteration. There are values that are passed from one iteration to the next through one or more other variables, here just `sum`. These are global to the body of the loop in order to persist from one iteration to another. So the iterations are communicating with each other via side-effects.

Both of these features are anathema to functional programming. You can't have variables changing from one iteration to the next like that. So `for` and `while` loops and anything similar is out of the question.

Instead in functional programming, loops are done with recursion within a function. The idea is that the computation within the recursive function represents the computation of one iteration of the loop. The next iteration is

started by calling the function again. Values are communicated between two iterations via the arguments passed to the function. The rebinding of the argument variables to different values on different calls is the closest you will get to changing the value of a variable in pure functional programming.

Here is the same function in SML. In the first version I've written the case expression out in full.

```
fun sumlist the_list =
(
    case the_list of
      []        => 0
    | (v::rest) => v + (sumlist rest)
)
```

It can be made neater by merging the case expression with the function definition.

```
fun sumlist []        = 0
|   sumlist (v::rest) = v + (sumlist rest)
```

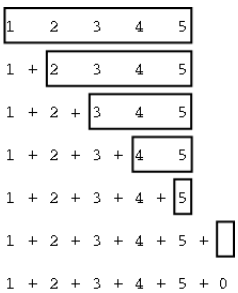
The algorithm is described by the following specification.

To find the sum of the elements in a list:

1. If the list is empty then define the sum to be 0.
2. If the list is not empty then we can split it into a first element and some remaining elements. The sum is the value of the first element plus the sum of the remaining elements.

This is a "divide and conquer" style of specification. We reduce the problem of summing the list to a smaller problem by splitting it into two pieces. SML provides a cheap mechanism to split the first element off from a list. Figure 2-2 shows this division.

Figure 2-2. Summing a list by Divide and Conquer.



To find the sum of the list `[1, 2, 3, 4, 5]` we reduce it to the problem of finding the sum of the list `[2, 3, 4, 5]` and adding 1 to it. This continues until we get to an empty list in which case its sum is known trivially. Then we can complete the additions.

The problem with this algorithm is that the addition cannot be completed until the sum of the remaining elements is known. This leaves a trail of pending additions which is as long as the list itself. Each pending addition means there is an incomplete function call taking up stack space. So the stack space consumed is proportional to the length of the list. The iterative algorithm in C above takes up constant stack space. When imperative programmers point to the superiority of iterative loops this is a major complaint they make against recursion.

But this problem is not a problem with recursion itself as much as how it is implemented in imperative languages like C. We can sum the list with recursion in constant stack space too using tail recursion.

Tail Recursion

A tail call to a function is one that is the last step made in the execution of the calling function. In other words, the return value of the calling function will *unconditionally* be the return value of the called function. Since there

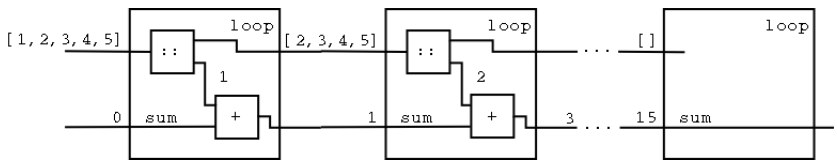
will be no more execution in the calling function its stack space can be reclaimed *before* the tail call. This eliminates the accumulation of stack space during the loop. This is called the tail call optimisation.

Here is the same function in SML taking care to use tail recursion.

```
fun sumlist the_list =
let
    fun loop []        sum = sum
    |   loop (v::rest) sum = loop rest (sum+v)
in
    loop the_list 0
end
```

The first argument to the `loop` function is the remainder of the list to be counted. The `sum` variable accumulates the result. The initial call to `loop` supplies initial values for these variables. Each subsequent call to the function passes updated values for these variables. When the remainder of the list is empty then the value of the `sum` variable is the number of list elements. Figure 2-3 shows the function calls in a data flow diagram.

Figure 2-3. Tail Recursion as Data Flow



Each iteration of the loop function is an operation that shortens the list by one and increments the sum. When the list is reduced to an empty list then the accumulated sum is the answer.

I emphasised the word unconditionally in the definition of tail calls above because sometimes what is a tail call can be obscured. For example if there is an exception handler surrounding the tail call then this implies that the calling function may sometimes still have work to do handling an exception

from the called function so it can't be a tail call. You may need to watch out for this in loops.

Tail Recursion as Iteration

The first reaction of a C programmer to using recursion everywhere is to object to the performance of using recursion instead of iteration. But the programmer's intuition on what is expensive and what isn't is based on how C works and is pretty much useless when it comes to functional programming because it works so differently.

In this section I want to emphasise the equivalence of tail recursion and iteration by working back from recursion to iteration. This will not only show that tail recursion is as efficient as iteration but will also provide an intuition that will be useful for bringing across imperative idioms to functional programming.

If you have studied a little assembly language you will know of the variety of machine instructions for jumping around in a program. The Intel architecture has the simple unconditional `JMP` instruction. This corresponds to the `goto` statement in C. You would expect that a `goto` translates to a single `JMP` instruction. For calling functions there is the `CALL` instruction which works like `JMP` except that it saves a return address on the stack. This allows the calling function to continue execution after the called function has finished.

But when we have tail recursion there is nothing to return to. By definition the calling function has completed. So instead of using a `CALL`, we should be able to use a `JMP` instruction to implement the tail call. In other words, *a tail call is equivalent to a `goto`*.

I'll demonstrate this equivalence by manually translating the `sumlist` function to C. Here is the original in SML.

```
fun sumlist the_list =
let
```

```
    fun loop []        sum = sum
    |   loop (v::rest) sum = loop rest (sum+v)
in
    loop the_list 0
end
```

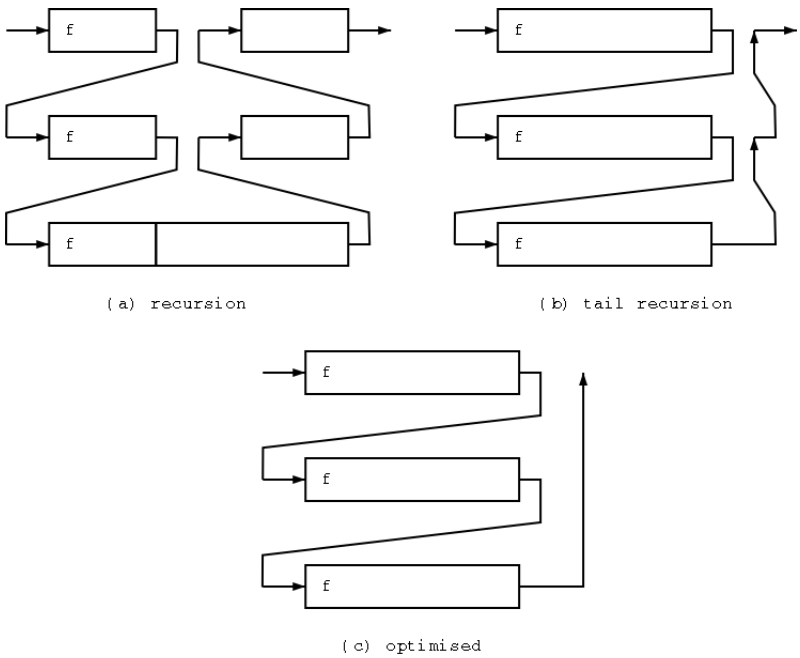
In C, using the `Node` type in the section called *The Basics*, I get the (somewhat literal) code:

```
int
sumlist(const Node* the_list)
{
    const Node* list;       /* args to loop */
    int         sum;

    list = the_list;        /* args to the first call */
    sum  = 0;
    goto loop;              /* a tail call to loop */

loop:
    if (list == 0)
    {
        return sum;         /* value returned from loop */
    }
    else
    {
        int         v    = list->value;
        const Node* rest = list->next;

        list = rest;        /* new args for the tail call */
        sum  = sum + v;
        goto loop;
    }
}
```

Since all calls to `loop` are tail calls I can use `goto`s instead of function calls and use a label for `loop`. This translation simultaneously incorporates the tail call optimisation and the inlining of the `loop` function. A good SML compiler can be expected to perform these optimisations as a matter of course and generate machine code as good as C for the `sumlist` function.

Just to belabor the point, Figure 2-4 shows the equivalence graphically.

Figure 2-4. Tail Recursion as Iteration



Part (a) of the figure shows a function `f` in a recursive loop being called. In the middle of its execution it calls itself recursively. This continues until one of the invocations chooses not to call itself. Then the invocation returns and the second half of the previous invocation executes. This continues until all invocations have returned. Part (b) shows what we have with tail recursion. There is no second half, the returned value from one invocation becomes the returned value from the previous one and eventually the returned value from the entire function. Looking at the diagram we see that the cascade of returns is redundant. In part (c) the last invocation returns directly for the whole loop. With a bit of inlining the recursion has become just a sequence of executions of the body of the function `f` joined by `goto` statements, in other words conventional imperative iteration.

Using the Fold Functions

The structure of the code in Figure 2-3 is such a common pattern that there is a standard built-in function to implement it. It is called `List.foldl`, but it is also directly callable as `foldl`[2]. Actually there are two variants, `foldl` and `foldr`, depending on whether you want to read the list from left to right or right to left. Normally you should read the list left to right for efficiency.

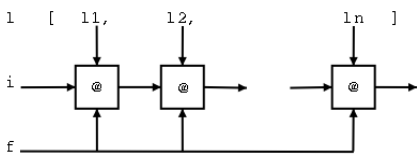
The function for summing the list using `foldl` can now be written

```
fun sumlist the_list = foldl (fn (v, sum) => v + sum) 0 the_list
```

The first argument to `foldl` is a function that performs the body of the loop. It takes a pair of arguments, the first is a value from the list and the second is the accumulated value. It must then compute a new accumulated value. The second argument to `foldl` is the initial value for the accumulator and the third is the list. The `foldl` takes care of all of the iteration over the list.

In general the expression `foldl f i l` corresponds to the data flow diagram in Figure 2-5. In this diagram I have represented the calling of the function `f` by an @ operator. This applies the function to the pair of the list element and the accumulated value. These two values are always supplied in a single argument as a tuple with the list element first and the accumulated value second.

Figure 2-5. The Data Flow for foldl



There are further abbreviations you can do in the `foldl` call. A function that just adds two integers together can be derived directly from the addition operator.

```
fun sumlist the_list = foldl (op +) 0 the_list
```

The notation `(op +)` makes the addition operator into a function that can be passed around like any other. The type of the function is declared in the standard `INTEGER` signature as `(int * int) -> int` which means it takes a pair of integers as its argument, just as needed by `foldl`.

This notation will only work if the compiler can work out from the context that it is the integer addition operator that is needed, rather than real or some other type. It can do this in this case because the initial value is known to be the integer zero. If you wrote `foldl (op +) 0.0 the_list` then it would know to use the real addition operator. You can't write a sum function that can sum either lists of integers or lists of reals.

The order of the arguments to `foldl` is meaningful. You can use *currying* to omit arguments from the right to create partially satisfied functions. For example the expression `foldl (op +) 0` represents a function that will take a list and return its sum. You can write

```
val sumlist = foldl (op +) 0
```

which binds the name `sumlist` to this partially satisified function. When you write `sumlist [1, 2, 3]` you satisfy all of the arguments for the `foldl` and it executes. Similarly you could define

```
val accumlist = foldl (op +)
val sumlist = accumlist 0
```

and if you wrote `accumlist x [1, 2, 3]` you would be accumulating the sum of the list elements onto the value of `x`. (The compiler will default `accumlist` to do integer addition in the absence of any type constraints saying otherwise).

As a general rule when choosing the order of arguments, if you want to make currying useful, then place the argument that varies the least first and the most varying last. The designers of `foldl` judged that you are more likely to want to apply the same function to a variety of lists than apply a variety of functions to a particular list. You can think of the first arguments as customisation arguments so that `foldl (op +) 0` customises `foldl` to

sum lists as opposed to `foldl (op *) 1` which multiplies list elements together.

Tail Recursion for Finite State Machines

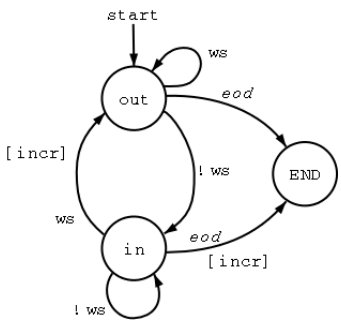
A finite state machine, or FSM, is a common design technique for describing repetitive behaviour. The FSM passes through a series of discrete states in response to its inputs. As it makes the transition from one state to another it performs some output action. This may continue forever or there may be an end state. The word finite in the name refers to the finite number of different discrete states that the machine can be in, not how long the machine runs for.

Figure 2-6 shows a FSM to count words in text. There are two operational states: `in` means the machine is inside a word, `out` means the machine is outside a word. The `END` state stops the machine. The text is supplied as a sequence of characters. Each character causes a transition to another state, which may be the same as the previous state.

If the machine is in the `out` state and it gets a white space character, represented by `ws` in the figure, then it stays in the `out` state. If it gets a non-white space character (and it's not the end of data, `eod`) then it changes to the `in` state because it has entered a word. A word is completed when there is a transition from the `in` state back to the `out` state upon a white space character. The `[incr]` notation means that the count of words is incremented during the transition.

If the machine gets an end-of-data condition then it stops at the `END` state. A word has to be counted if it is was in a word at the time.

Figure 2-6. Counting Words with a FSM



If you were to write this FSM in C you might implement the states with small pieces of code joined with `goto` statements. It might be something like:

```
int
word_count(const char* text)
{
    int     count = 0;
    char    c;

out:
    c = *text++;
    if (!c) goto eod;
    if (!isspace(c)) goto in;
    goto out;

in:
    c = *text++;
    if (!c)
    {
        count++
        goto eod;
    }
    if (isspace(c))
    {
        count++;
        goto in;
    }
    goto in;

eod:
```

```
    return count;
}
```

(This is a bit ugly but it's a literal translation of the design and it should generate nice fast machine code if you care.)

Now that we know that tail recursion in functional programming is equivalent to `goto` in imperative programming we can write the same algorithm directly in SML. The set of states will correspond to a set of mutually tail-recursive functions. Here is the word counter function.

```
and word_count text =
let
    fun out_state []        count = count
    |   out_state (c::rest) count =
    (
        if Char.isSpace c
        then
            out_state rest count
        else
            in_state rest count
    )

    and in_state []        count = count + 1
    |   in_state (c::rest) count =
    (
        if Char.isSpace c
        then
            out_state rest (count + 1)
        else
            in_state rest count
    )
in
    out_state (explode text) 0
end
```

The two state functions are part of a mutually recursive pair joined by the `and` keyword. For convenience I've represented the text as a list of characters. The built-in `explode` function makes a list of characters from a string and the built-in `Char.isSpace` tests if the character is white space. The output from the loop is in the accumulator `count`. It gets incremented

whenever we leave the `in` state. In place of an explicit `eod` state we just return the accumulated count.

Here is the main function that calls the `word_count` function.

```
fun main(arg0, argv) =
let
    val cnt = word_count "the quick brown fox";
in
    print(concat["Count = ", Int.toString cnt, "\n"]);
    OS.Process.success
end
```

It counts the word in the foxy message and prints the result. To print I've used the built-in `concat` function which concatenates a list of strings into a single string and `Int.toString` to make a string from an integer.

Alternatively you can represent the state by a state variable and have a single loop. The `word_count` function then becomes:

```
and word_count text =
let
    datatype State = In | Out

    fun loop Out []        count = count

    |   loop Out (c::rest) count =
    (
        if Char.isSpace c
        then
            loop Out rest count
        else
            loop In rest count
    )

    |   loop In []        count = count + 1

    |   loop In (c::rest) count =
    (
        if Char.isSpace c
        then
            loop Out rest (count + 1)
        else
            loop In rest count
    )
```

```
in
    loop Out (explode text) 0
end
```

In this code I've used a datatype to define some states. This use of a datatype is equivalent to an enumeration type in C or C++. Apart from that there is little difference in such a small example. We now have just one tail-recursive function that takes a current state argument. I've used pattern matching in the function definition to recognise all of the combinations of state and character list.

Concerning performance, using `explode` requires copying the string to a list before counting. If you were dealing with long strings and were worried about the amount of memory needed for two copies then you could try just subscripting the string using `String.sub`. This may well run slower though since there is bounds-checking on each subscript call.

This word counting example is just an example to demonstrate a simple state machine. It's a bit of overkill. The shortest piece of code to count words in a string uses `String.tokens`:

```
fun word_count text = length(String.tokens Char.isSpace text)
```

The *getopt* Programs

The *getopt* program will fetch the options from the command line and check that the required options are present. I'll implement this in several different ways to demonstrate some programming techniques.

The goal is to to recognise the following options.

-v
–verbose

This is optional.

–width *width*

This is required. It sets the width.

–height *height*

This is required. It sets the height.

-h

This prints a help message.

files

The remaining arguments are file names.

Our little program will print the file names. If the verbose option is given then it will also print the width and height. The usage will be:

```
Usage: [-h] [-v|-verbose]
       [-width width]
       [-height height]
       files
```

Mostly Functional

This first version, `getopt1.sml`, is in a mostly-functional style. The deviation is in the use of an exception to abort the program with an error message.

The first part of the program has some type definitions for documentation.[3]

```
(*  The options will be returned as a list of pairs
    of name and value.  We need to use an option type for
    the value so that we can distinguish between a missing
    value and an empty value.

*)
type Option = string * string option

(*  The result from the command line parsing will
    be a list of file names and a set of options.
```

```
*)
type CmdLine = (Option list) * (string list)

(*  This exception will bomb with a usage message. *)
exception Usage of string
```

I've defined `Option` to be a pair of a string for the name of the option and an optional string for its value. The name will be an internal canonical name. The `CmdLine` type is to describe the result from `parse_cmdline`, namely a list of options and a list of files.

I've defined an exception `Usage` which carries a message. I use this to abort the program. The exception is caught when it aborts the `main` function and prints a message on `stdErr`. The exception handler returns the failure code so that the program exits with an exit code of 1.

The next section of the program scans the arguments.

```
fun parse_cmdline argv : CmdLine =
let
    fun loop [] opts = (opts, [])       (* no more args *)

    |   loop ("-h"::rest) opts =
            loop rest (("help", NONE) :: opts)

    |   loop ("-v"::rest)        opts =
            loop rest (("verbose", NONE) :: opts)

    |   loop ("-verbose"::rest) opts =
            loop rest (("verbose", NONE) :: opts)

    |   loop ("-width"::rest)  opts =
            get_value "width"  rest opts

    |   loop ("-height"::rest) opts =
            get_value "height" rest opts

    |   loop (arg::rest) opts =
    (
        if String.sub(arg, 0) = #"-"
        then
            raise Usage (concat["The option ", arg,
                            " is unrecognised."])
        else
```

```
            (opts, arg::rest)           (* the final result *)
    )

    and get_value name [] opts =
    (
        raise Usage (concat[
            "The value for the option ", name, " is missing."])
    )

    |   get_value name (v::rest) opts =
    (
        loop rest ((name, SOME v) :: opts)
    )
in
    loop argv []
end
```

The `parse_cmdline` function scans the arguments in an inner loop. I've got a type constraint on the expression pattern `parse_cmdline argv` to indicate that its resulting value is of the type `CmdLine`. Although this is not strictly necessary it aids readability. It can also make it easier to find the location of type errors by putting in explicit points where the type is known. (See Appendix B for a discussion on type errors). If you wanted a type constraint on the `argv` argument then you would need to put it in parentheses i.e. `(argv: string list)`.

I've used literal strings in the binding patterns for conciseness. So for example the second variant of the loop function says that if the argument list starts with a `"-h"` then continue looping over the rest of the arguments with the options table in `opts` augmented with the pair `("help", NONE)`.

The first variant catches the case of running out of arguments while scanning for options. In this case I return the options that I have and the list of files is empty.

To handle an option which requires a value I've used a separate function, `get_value`. It looks at the first of the rest of the arguments. If the rest of the arguments are empty then the value is missing. If present then I add it to the `opts` table and continue the loop. Note that with an `option` type, a value that is present is tagged with the `SOME` data constructor. This

algorithm will treat the case of `-width -height` as a width option having the value `-height`.

The `get_value` function must be joined to the `loop` function with the `and` keyword to make the forward reference to `get_value` in `loop` legal. The two are mutually recursive functions.

The last variant of the loop function catches all the arguments that don't match any of the preceding option strings. Remember that in cases the variants are matched in order from first to last. An identifier in a binding pattern, here `arg`, will match any value. I need to check if the value starts with a hyphen in which case it is an unrecognised option. I've used the `String.sub` function which subscripts a string to get a character. The first character is at index 0. The `#"-"` notation is the hyphen character. If the argument does not start with a hyphen then I return the final result which is the options table and the rest of the arguments, not forgetting that `arg` is one of them.

I've used parentheses to bracket the body of each variant although in this code they are redundant. I find that I get fewer syntax surprises this way as the code gets more complex. Imagine a body containing a case expression!

The next section of the program has some utility functions to deal with option tables.

```
and find_option opts name : (string option) option =
(
    case List.find (fn (n, v) => n = name) opts of
      NONE => NONE
    | SOME (n, v) => SOME v
)


and has_option opts name =
    (find_option opts name) <> NONE


and require_option opts name and_value : string =
(
    case find_option opts name of
      NONE => raise Usage (concat[
```

```
                    "The option '", name,
                    "' is missing."])

    | SOME NONE =>  (* found but has no value *)
    (
        if and_value
        then
            raise Usage (concat[
                "The option '", name, "' is missing a value."])
        else
            ""
    )

    | SOME (SOME v) => v  (* found and has a value *)
)
```

The `find_option` searches the table for a given name. I've used the `List.find` function which finds the first entry in the option list that satisfies the predicate function which is the first argument to `List.find`. Remember that each member of the list is a pair of name and value. So the argument to the predicate is a pair, matching with `(n, v)`. The predicate tests if the name field is the same as the supplied name.

The option tables have a subtle property. I built them in reverse by pushing new options onto the front of the list. So if there are duplicate options then the first one found will be the last on the command line. I either should not rely on this or I should document it *loudly*.

The result from the `List.find` will be of the type `Option option`. That is it will be `NONE` if the option was not found or else some name-value pair, `SOME (n, v)`. I've decided that I only want to return the value. But I have to indicate if the value was found or not in the options table so I wrap it in another level of `option`.

The `has_option` just tests if `find_option` returns a non-NONE value. The equality and inequality (<>) operators are available for the type `T option` if they are available for a type `T`.

The `require_option` function checks that an option is present in the table. If the `and_value` flag is true then I also require it to have a value. If it has a

value then I return it. Because every `if` expression must have both a `then` and an `else` part I can't avoid covering the case of an option not having a value and not needing to, even though I don't use this case in the program. Better safe than sorry.

The final part of the program is the main function. It should be fairly straightforward.

```
fun main(arg0, argv) =
let
    val (opts, files) = parse_cmdline argv

    val width  = require_option opts "width"  true
    val height = require_option opts "height" true

    fun show_stuff() =
    (
        print "The files are";
        app (fn f => (print " "; print f)) files;
        print ".\n";

        if has_option opts "verbose"
        then
            print(concat[
                "The width is ",  width,  ".\n",
                "The height is ", height, ".\n"
                ])
        else
            ()
    )
in
    if has_option opts "help"
    then
        print "some helpful blurb\n"
    else
        show_stuff();

    OS.Process.success
end
handle Usage msg =>
(
    TextIO.output(TextIO.stdErr, concat[msg,
       "\nUsage: [-h] [-v|-verbose] [-width width]",
       " [-height height] files\n"]);
    OS.Process.failure
```

```
)
```

Observe again how all `if` expressions must have both a `then` and an `else` part and each part must return a value. Since the `print` function has `unit` as a return value then the `else` part must too. The `()` is the notation for the one and only value of the `unit` type. You can interpret it as "do nothing".

In a real program of course I would call some function to do the work of the program and pass it the options and the file names. But the rest of the code of the program may be quite large and it will only refer to the options in a few places scattered throughout the program. It would be awkward to pass the options table all the way through the program just to be read in a few places. The program would quickly become difficult to read. Instead I will cheat and put the options into a global table. Since they are used read-only and are set before the body of the program is run this won't break referential transparency. The program will still be as good as pure.

The way to put values into global variables in SML is to use global reference values. Reference values emulate the variables of imperative programs. But we don't have to get our hands dirty dealing with them. The SML/NJ utility library includes a hash table module that uses reference values internally to store its contents imperatively. In the next section I show how to set up one of those.

Using a Hash Table

The SML/NJ utility library defines a generic hash table using imperative storage so that you can update its contents. (See Chapter 5). The table is generic over the key type. You need to supply a specification of the key type and its properties to make an instance of the table type. Then you can create values of the table type. All of the values have the same key type but each can have a different content type, since the table is polymorphic in this type. But all entries in a particular table have the same content type. SML does not do dynamic typing or subtyping.

The generic hash table is defined by this functor from the `hash-table-fn.sml` file in the SML/NJ library.

```
functor HashTableFn (Key : HASH_KEY) : MONO_HASH_TABLE
```

The functor takes a `Key` structure as an argument and produces a hash table structure. The `HASH_KEY` signature describes what the `Key` structure must tell the functor. Observe that in signatures, functions are described as a value of a function type.

```
signature HASH_KEY =
sig
  type hash_key

  val hashVal : hash_key -> word
      (* Compute an unsigned integer from a hash key. *)

  val sameKey : (hash_key * hash_key) -> bool
      (* Return true if two keys are the same.
       * NOTE: if sameKey(h1, h2), then it must be the
       * case that (hashVal h1 = hashVal h2).
       *)

end (* HASH_KEY *)
```

For the option table I want strings for keys. So I've defined a string table key with the following structure. The hash function comes from another library module in the `hash-string.sml` file.

```
structure STRT_key =
struct
    type hash_key = string
    val hashVal = HashString.hashString
    fun sameKey (s1, s2) = (s1 = s2)
end
```

Now I can assemble these to make a module I call `STRT` that implements a hash table from strings to some content type. I've also defined a useful exception that will be used later for when table lookups fail.

```
structure STRT = HashTableFn(STRT_key)
```

```
exception NotFound
```

This structure conforms to the MONO_HASH_TABLE signature. Here MONO means it is monomorphic in the key type. This signature describes all of the types and values (including functions) that the hash table structure makes public. Here is a part of this signature containing the features that I use often.

```
signature MONO_HASH_TABLE =
sig

  structure Key : HASH_KEY

  type 'a hash_table

  val mkTable: (int * exn) -> 'a hash_table
      (* Create a new table; the int is a size hint
       * and the exception is to be raised by find.
       *)

  val insert: 'a hash_table -> (Key.hash_key * 'a) -> unit
      (* Insert an item.  If the key already has an item
       * associated with it, then the old item is
       * discarded.
       *)

  val lookup: 'a hash_table -> Key.hash_key -> 'a
      (* Find an item, the table's exception is raised
       * if * the item doesn't exist
       *)

  val find: 'a hash_table -> Key.hash_key -> 'a option
      (* Look for an item, return NONE if the item
       * doesn't exist
       *)

  val listItemsi: 'a hash_table -> (Key.hash_key * 'a) list
      (* Return a list of the items (and their keys) in
       * the table
       *)
end
```

This shows that the table structure exports a copy of the `Key` structure that defined it. This is good practice as it can be useful to get access to the hash function of the table.

So now I have the type `'a STRT.hash_table` which maps from string keys to some content type represented by the type variable `'a`. I can create a table from strings to strings like this.

```
type OptionTable = string STRT.hash_table

val option_tbl: OptionTable = STRT.mkTable(101, NotFound)
```

The type constraint on the table value settles the type of the table immediately to save the compiler and the reader having to figure it out.

Getopt with a Hash Table

With these hash table tools I can go on to write a neater *getopt* program, called `getopt2.sml`. I'm in the habit of putting useful things like the string table structure into a common module which can be used throughout a project. I put global variables like the option table into their own separate module. These would normally go into separate files. In the source code for this program I've put them all in the same file. Here is the common module which exports all of its declarations.

```
structure Common =
struct

(*--------------------------------*)
(*  A hash table with string keys. *)

    structure STRT_key =
    struct
        type hash_key = string
        val hashVal = HashString.hashString
        fun sameKey (s1, s2) = (s1 = s2)
    end

    structure STRT = HashTableFn(STRT_key)
```

```
    exception NotFound

(*---------------------------------*)

end
```

Then I define a signature for the global module to constrain what it exports. It's got a basic API for setting and testing options. In keeping with the previous *getopt* program an option value is an optional string so that I can tell the difference between a missing option value and an empty option value.

```
signature GLOBAL =
sig
    type Option = string option

    (*  Add an option to the table silently overriding
        an existing entry.
    *)
    val addOption: (string * Option) -> unit

    (*  Test if an option is in the table. *)
    val hasOption: string -> bool

    (*  Get the value of an option if it exists. *)
    val getOption: string -> Option option

end
```

Next I define the global module. The `open` declaration imports everything from `Common` and makes its names directly visible. Note that there must be a definition for every name declared in the `GLOBAL` signature so the `Option` type must be defined again.

```
structure Global: GLOBAL =
struct
    open Common

(*---------------------------------*)
(*  The option table. *)

    type Option = string option
```

```
    type OptionTable = Option STRT.hash_table

    val option_tbl: OptionTable = STRT.mkTable(20, NotFound)

    fun addOption arg  = STRT.insert option_tbl arg

    fun hasOption name = STRT.find option_tbl name <> NONE

    fun getOption name = STRT.find option_tbl name

(*--------------------------------*)

end
```

The option table is a value in the structure. This value will be created when the module is compiled into the heap as I described in the section called *Assembling the Hello World Program*. The value comes from the `mkTable` function. It will end up in the exported heap file. When defining `addOption` I made the argument type match the argument to the STRT.insert function. This avoids unpacking and repacking the contents as it passes from function to function.

I could have abbreviated the definitions of `addOption` and `getOption` further by taking advantage of currying but I think that this obscures the code a bit for no real gain.

```
val addOption = STRT.insert option_tbl
val getOption = STRT.find   option_tbl
```

Finally the main program is rewritten to eliminate all mention of a table of options.

```
structure Main=
struct

    (* This exception will bomb with a usage message. *)
    exception Usage of string


    fun parse_cmdline argv : string list =
    let
        fun loop [] = []            (* no more arguments *)
```

```
    |   loop ("-h"::rest) = add ("help", NONE) rest

    |   loop ("-v"::rest) = add ("verbose", NONE) rest

    |   loop ("-verbose"::rest) = add ("verbose", NONE) rest

    |   loop ("-width"::rest)  = get_value "width"  rest

    |   loop ("-height"::rest) = get_value "height" rest

    |   loop (arg::rest) =
    (
        if String.sub(arg, 0) = #"-"
        then
            raise Usage (concat[
                    "The option ", arg, " is unrecognised."])
        else
            arg::rest           (* the final result *)
    )

    and get_value name [] =
    (
        raise Usage (concat["The value for the option ",
                            name, " is missing."])
    )
    |   get_value name (v::rest) = add (name, SOME v) rest

    and add pair rest =
    (
        Global.addOption pair;
        loop rest
    )
in
    loop argv
end


fun require_option name and_value : string =
(
    case Global.getOption name of
      NONE => raise Usage (concat[
                "The option '", name, "' is missing."])

    | SOME NONE =>              (* found but no value *)
    (
        if and_value
```

```
        then
            raise Usage (concat["The option '", name,
                                "' is missing a value."])
        else
            ""
    )

    | SOME (SOME v) => v        (* found with a value *)
)



fun main(arg0, argv) =
let
    val files = parse_cmdline argv

    val width  = require_option "width"  true
    val height = require_option "height" true

    fun show_stuff() =
    (
        print "The files are";
        app (fn f => (print " "; print f)) files;
        print ".\n";

        if Global.hasOption "verbose"
        then
            print(concat[
                "The width is ",  width,  ".\n",
                "The height is ", height, ".\n"
                ])
        else
            ()
    )
in
    if Global.hasOption "help"
    then
        print "some helpful blurb\n"
    else
        show_stuff();

    OS.Process.success
end
handle Usage msg =>
(
    TextIO.output(TextIO.stdErr, concat[msg,
     "\nUsage: [-h] [-v|-verbose] [-width width]",
```

```
         " [-height height] files\n"]);
        OS.Process.failure
    )


    val _ = SMLofNJ.exportFn("getopt2", main)
end
```

Since I am now using modules from the SML/NJ utility library I must mention the library in the CM file for the program. Here is the `getopt2.cm` file. It has the path to a CM file for the library which was created when SML/NJ was installed.

```
group is
    getopt2.sml
    /src/smlnj/current/lib/smlnj-lib.cm
```

An alternative to having the table in the heap is to have it built on demand when the program runs. A convenient way to do this in SML/NJ is described in the section called *Lazy Suspensions* in Chapter 4.

The Deluxe getopt

The *getopt* programs I've done so far implement a simple command line syntax. This next one, called `getopt3.sml`, does the full Gnu-style syntax with short and long options etc. It will report its usage as:

```
Usage: getopt
  -v  -verbose        Select verbose output
      -width=width    The width in pixels
      -height=height  The height in pixels
  -h  -help           Show this message.
```

I've written this program using the `GetOpt` structure in the SML/NJ utility library. This structure is rather under-documented and not that easy to figure out. You can find its signature in the `getopt-sig.sml` file. When you use this module to parse the command line you get back a list of options and

files similar to my first *getopt* program. But I will then transfer them to a global option table as in the the second *getopt* program.

I start by building an `Option` module that contains the command line parsing and can deliver the values of the options imperatively. The API for this module is specified in the `OPTION` signature. I've put in an alias of `G` for the `GetOpt` structure to save on typing.

```
structure Option: OPTION =
struct
    structure G = GetOpt
```

The interface to the `GetOpt` structure revolves around a single type to represent all of the possible options. This should be a datatype to be useful. I start by defining the `Option` type. I keep the width and height as strings for simplicity but in a real program you would probably use integers with `Int.fromString` to do the conversion.

```
(*  This represents an option found on
    the command line.
*)
datatype Option =
        Verbose
    |   Help
    |   Width of string
    |   Height of string
```

An option is described by the following record type in `GetOpt`.

```
type 'a opt_descr = {
    short : string,
    long : string list,
    desc : 'a arg_descr,
    help : string
    }
    (* Description of a single option *)
```

The `short` field contains the single letter version of the option. If you have more than one letter then they are treated as synonyms. The long field contains the `long` version of the option as a list of strings. Again if you have more than one then they are treated as synonyms.

The `desc` field describes properties of the options value and how to map it to the representation type (my `Option`). The value of this field is of the following datatype from `GetOpt`.

```
datatype 'a arg_descr
    = NoArg of unit -> 'a
    | ReqArg of (string -> 'a) * string
    | OptArg of (string option -> 'a) * string
```

The `'a` type variable is a place holder for the representation type. If the option takes no value then supply `NoArg`. You must include with it a function that returns a representation value for the option. If the option requires a value then supply `ReqArg` along with a function to convert the value to the representation type and a description of the value for the usage message. If the option's value is optional then supply `OptArg` along with a conversion function and a description as for `ReqArg`. Note that for `OptArg` the conversion function is passed a `string option` type to tell whether the value is available or not.

Here is my code for part of the option description list.

```
fun NoArg  opt       = G.NoArg (fn () => opt)
fun ReqArg opt descr = G.ReqArg (opt, descr)

val options: (Option G.opt_descr) list = [
        {short = "v", long = ["verbose"],
            desc = NoArg Verbose,
            help = "Select verbose output"
        },

        {short = "", long = ["width"],
            desc = ReqArg Width "width",
            help = "The width in pixels"
        },
```

I've defined two helper functions to make it easier to write the option descriptions. My `NoArg` function takes a value of the representation type and wraps it into a conversion function for the `GetOpt.NoArg` data constructor.

My `ReqArg` does a similar thing but here the first argument `opt` is the conversion function. The data constructors `Width` and `Height` in the

`Option` type can be used as functions to construct values of type `Option` from the types that they tag. For example the `Width` data constructor behaves as a function with the type `string -> Option` which is just the type needed for `GetOpt.ReqArg`.

The command line is parsed using the GetOpt.getOpt function. Its signature is

```
datatype 'a arg_order
  = RequireOrder
  | Permute
  | ReturnInOrder of string -> 'a
(* What to do with options following non-options:
 * RequireOrder: no processing after first non-option
 * Permute: freely intersperse options and non-options
 * ReturnInOrder: wrap non-options into options
 *)

val getOpt : {
        argOrder : 'a arg_order,
        options : 'a opt_descr list,
        errFn : string -> unit
      } -> string list -> ('a list * string list)

(* takes as argument an arg_order to specify the
 * non-options handling, a list of option descriptions
 * and a command line containing the options and
 * arguments, and returns a list of (options,
 * non-options)
 *)
```

The first argument is a record of details about the options. I'll just use `RequireOrder` for the ordering control. The `options` field is my list of option descriptions. For an error function I just need something to print a string to `stdErr`.

```
    fun toErr msg = TextIO.output(TextIO.stdErr, msg)
```

The second argument is the argv list. The result is a list of representation values and the remaining arguments, the file names. Here is my code to parse the command line.

```
val opt_tbl: (Option list) ref = ref []


fun parseCmdLine argv =
let
    val (opts, files) =
        G.getOpt {
            argOrder = G.RequireOrder,
            options  = options,
            errFn    = toErr
            } argv
in
    opt_tbl := opts;
    files
end
```

When I get back the list of options I assign it to the imperative variable `opt_tbl`. The variable must have an initial value which is constructed by the `ref` data constructor from an empty list.

Then I can write some accessor functions to get option information from the table.

```
fun hasVerbose() =
(
    List.exists (fn opt => opt = Verbose) (!opt_tbl)
)

fun hasHelp() =
(
    List.exists (fn opt => opt = Help) (!opt_tbl)
)

fun getWidth()   =
let
    val opt_width = List.find
                    (fn Width _ => true | _ => false)
                    (!opt_tbl)
in
    case opt_width of
      NONE          => NONE
    | SOME(Width w) => SOME w
    | _             => raise Fail "Option,getWidth"
)
```

The `!` operator dereferences the imperative variable. It is like the `*` operator in C. The operator precedence rules require the parentheses around the dereference. I've used `List.exists` to check for the presence of the simple options in the table.

To get the width value out I need `List.find` to return the entry from the list. The predicate needs some more elaborate binding patterns in order to recognise the `Width` tag and ignore the string value with an underscore. The case expression must cover all possibilities or else you will get messy warnings from the compiler which should be avoided at all costs or some day you will miss a genuine error among the warnings. In this example I need to cover the find returning a non-`Width` entry even though that is impossible for the predicate I've used. The `Fail` exception is a built-in exception in the Basis library that you can use to signal an internal fatal error. Use it only for impossible conditions like this. The `GetOpt` implementation also uses it for impossible conditions.

Finally I rewrite the `require_option` function to check that the option was supplied on the command line. It works for all of the `get*` functions.

```
fun require_option func name : string =
(
    case func() of
      NONE => raise Usage (concat[
                        "The option '", name,
                        "' is missing."])

    | SOME v => v
)
```

Then my main function becomes:

```
fun main(arg0, argv) =
let
    val files  = Option.parseCmdLine argv

    val width  = require_option
                     Option.getWidth "width"

    val height = require_option
                     Option.getHeight "height"
```

```
    fun show_stuff() =
    (
        print "The files are";
        app (fn f => (print " "; print f)) files;
        print ".\n";

        if Option.hasVerbose()
        then
            print(concat[
                "The width is ",  width,  ".\n",
                "The height is ", height, ".\n"
                ])
        else
            ()
    )
in
    if Option.hasHelp()
    then
        print "some helpful blurb\n"
    else
        show_stuff();

    OS.Process.success
end
handle Usage msg =>
(
    toErr msg;              toErr "\n";
    toErr(Option.usage());  toErr "\n";

    OS.Process.failure
)
```

Notes

1. The details of specifying the CM file name will change in a future release of SML/NJ.
2. This function is called `reduce` in some languages

3. My naming convention dresses up public names in mixed case and uses lower case with underscores for private names. Types start with an uppercase.

Chapter 3. The Basis Library

The Basis library is the library of standard types, functions and modules that are built-in for the Standard ML language. There is a set of reference documentation in the form of HTML pages for the library. You can download this documentation from the SML/NJ home page[*SML*]. You will need a copy of this documentation to follow the programming examples in this book.

In this chapter I will give you a brief overview of the library and discuss in more detail some of the more obscure features of the library. At the time of writing this, a more detailed book on the library is still in preparation (by another author).

You should also have a copy of the source code of the SML/NJ compiler. The Appendix C describes how to download it. The compiler is itself written in SML. The only part you need to pay attention to is the `boot/` directory. This contains the source code of the Basis library as well as the SML/NJ extensions. I find it very useful to consult this when the documentation is unclear.

Preliminaries

Start by reading the Introduction to the library in the Basis documentation all the way through. Then read the "SML'96 Changes" section. This documents changes to the language since some of the books and tutorials on SML were written. The most notable changes for system programming are the addition of:

- the `char` type and character literals (previously characters were represented by strings of length 1);
- unsigned integer types called `word` with sizes of 8, 31 and 32, and the `0w` prefix for their literals;

- hexadecimal signed and unsigned integers.

The value polymorphism restriction is not something you will likely encounter. For the purposes of this book it mainly requires you to ensure that imperative variables using the `ref` type are restricted to contain a specific declared type.

The section on type abbreviations in signatures allows type declarations like the type `Option` in the section called *Getopt with a Hash Table* in Chapter 2. I won't pay any attention to opaque signature matching but you will see it in some of the library source where a signature is matched using "`:>`" instead of "`:`". The difference is that the opaque form ensures that a type can be exported from a module with absolutely no information about its implementation being available. Normally the compiler peeks inside modules and can recognise when different type names exported from different modules are the same type because they have the same representation inside the module.

The next section on the "Top Level Environment" lists all of the types and functions that are available directly without having to be qualified by a module name. The phrase "top level" refers to the level of the prompt you get when you start up the SML compiler. All of these top-level names are associated with one of the library modules and are documented in more detail there. This section just provides are reference for what names don't need module qualification. The overloading of operators and their precedence is also presented.

General

The `General` structure contains miscellaneous definitions.

The `exnName` and `exnMessage` are important for dealing with uncaught exceptions in your programs. You should always have some exception handlers around a top-level function looking something like this.

```
val success = OS.Process.success
val failure = OS.Process.failure


fun run args =
let
    ...
in
    process args;
    success
end
handle
  IO.Io {name, function, cause} =>
    (
        toErr(concat["IO Error ", name,
              " ", function,
              " ", exnMessage cause, "\n"]);
        failure
    )

| Fatal => (toErr "Aborting\n"; failure)

| InternalError msg =>
    (
        toErr(concat["Internal error, ", msg, "\n"]);
        failure
    )

(* misc exception *)
| x =>
    (
        toErr(concat["Uncaught exception: ", exnMessage x,"\n"]);
        failure
    )
```

I usually have a `Fatal` exception to abort a program upon a fatal error. Each site that detects the error produces an error message and raises `Fatal` to abort the program. I also have an `InternalError` exception to report all places that should be unreachabled. Each site that raises `InternalError` must include a message that uniquely identifiers the site for debugging. The last handler above catches all other exceptions and describes them.

The functions `!` and `:=` for dereferencing and assignment of imperative variables are defined here. Assignment is made infix with a precedence in

the top-level environment. Dereference is an ordinary prefix function and so you will normally need to use parentheses around it. For example `f !x` is interpreted as a call to the function `f` with two arguments, `!` and `x` when you probably meant `f (!x)`.

The function `o` composes two functions together. It is sometimes useful when passing a combination of functions as an argument. For example a function `map (lookup o uppercase)` could be applied to a list of strings to replace each one by converting it to uppercase and then passing it through a lookup function.

The before function looks a bit odd. You can use it to attach tracing messages to expressions. For example

```
let
    ....
in
    print "starting f\n";
    f x before print "done f\n"
end
```

Option

The `Option` structure provides some useful functions for dealing with optional values. For any value `v` the expression `(SOME v)` represents the value being present and `NONE` for an absent value.

When a function returns an optional value you can use `getOpt` to reduce it to a definite value. It does this by providing a default if the optional value is omitted. For example to look up a name in a translation table and return the original name if it isn't present write something like `getOpt(STRT.find table name, name)`.

If you are 100% confident that the optional result from a function is present you can use `valOf` to reduce it to a definite value. It will raise the `Option` exception if you are wrong.

The isSome function can be useful to detect if an optional value is present. If the base type of the option is an equality type you could write `(f x) = NONE`. But this won't work for all types for example, at the top level:

```
- val z = SOME (fn x => x);
val z = SOME fn : ('a -> 'a) option
- z = NONE;
stdIn:25.1-25.9 Error: operator and operand
don't agree [equality type required]
  operator domain: ''Z * ''Z
  operand:         ('Y -> 'Y) option * 'X option
  in expression:
    z = NONE
- Option.isSome z;
val it = true : bool
```

Here `z` is an optional function and you can't test two functions to see if they are the same so you can't write `z = NONE` either. You must either use `isSome` or use a case expression.

The other functions in the `Option` structure might be useful in special circumstances. The `map` function looks to be the most useful.

Bool

The `Bool` structure provides some useful functions for I/O with booleans. For example write

```
print(concat["The flag is ", Bool.toString flag, "\n"])}
```

to print out the value of a flag.

The `fromString` function will do the reverse but it returns a `bool option` to indicate errors. The `scan` function is for use with with the more general string scanning system in `StringCvt` which I will discuss in the section called *Text Scanning*.

Text

The Types

The `String` structure provides a typical collection of string operations. The documentation is straightforward.

Note that all subscripts are range checked. This can be a big performance hit if you use, for example, the `sub` function to search through a string. Where possible you should use some of the other functions. Alternatively you may find that using `explode` followed by list operations is faster, at the cost of using more memory. If you really need to do a lot of fast indexing into a string you can find a subscript function without range checking in the `Unsafe.CharVector.sub` function, which you can find in the `boot/Unsafe` directory of the compiler source.

The current version of SML/NJ (110.0.7) does not implement wide strings.

The `Char` structure provides a typical collection of character operations, again straightforward. The character classification functions like `isSpace` and case conversion are here.

Here are some examples.

```
fun uppercase s = String.map Char.toUpper s

(*  Apply HTML quoting. *)
fun quoteHTML v =
let
    fun quote #"\"" = "&quot;"
    |   quote #"&"  = "&amp;"
    |   quote #"<"  = "&lt;"
    |   quote c     = str c
in
    String.translate quote v
end

(*   Break a string into words at white space. *)
fun split s = String.tokens Char.isSpace s
```

For more elaborate string parsing on large strings the functions in the `Substring` structure will be more efficient. A substring is represented as a pointer to a range of characters in an underlying string. So you can work on pieces of the string without any copying. Here are some examples.

```
fun skipWhiteSpace s =
    Substring.string(
        Substring.dropl Char.isSpace (Substring.all s))

fun countLines s =
    Substring.foldl
        (fn (ch,n) => if ch = #"\n" then n+1 else n) 1
        (Substring.all s)
```

Text Scanning

The `StringCvt` structure provides the infrastructure for reading values out of text. The infrastructure is based around the idea of a reader function that can split a value off of the beginning of a stream. Then there are transformers that build up more complex readers from simpler readers.

You have complete freedom in how you represent and implement streams just as long as they have a functional style. This means that if you have a stream `strm` and a `get` function that gets the first value of the stream then the expression `(get strm)` can be evaluated as many times as you like and it will always return the same value since it has no side-effects on the stream. The type of a reader function is

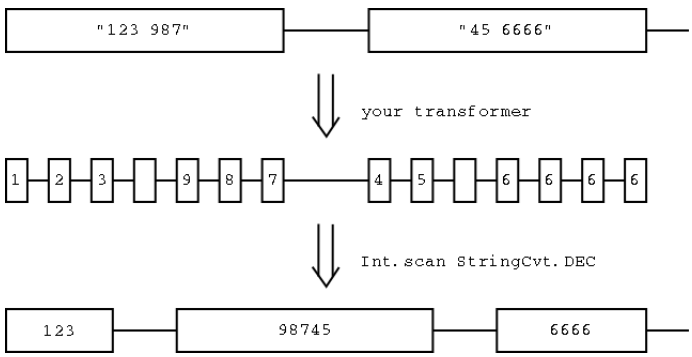
```
type (’a, ’b) reader = ’b -> (’a * ’b) option
```

In this, type `'b` represents the stream and `'a` is a value from the stream. The reader function returns one value from the front of the stream along with the remainder of the stream. An option type is used with NONE representing failure to find a value in the stream. So for example if `strm` contains the characters `"abc"` and `get` is a character reader then `(get`

`strm`) returns the first character from the stream along with the rest of the stream as the pair `SOME(#"a", "bc")`.

Transformer functions convert a stream of some type T to a stream of another type U. In the infrastructure they are applied on the fly. You have to at some point arrive at a stream of characters for the text scanning. Figure 3-1 shows an example of transformations for reading a stream of integers from character buffers. You write your transformer that can deliver characters one by one from the buffers. This will involve maintaining an offset into the buffer for where the next character will come from. Then you can use the `Int.scan` function to transform this stream of characters into a stream of integers.

Figure 3-1. Some Stream Transformations



Some possible sources of character streams are:

- a string. The `StringCvt.scanString` will transform a string to a character stream and deliver it to a transformer, and return the first value from the transformed stream. It is designed for one-off use to implement `fromString` functions.
- an input stream from a file. The `TextIO.scanStream` function will deliver a stream of characters from a file to a transformer and then return

the first value from the transformed stream. Since I/O streams are imperative the stream will be updated but it will be pure enough to complete the reading of the value.

- a list of characters. The `List.getItems` function can deliver list elements matching the requirements of a reader.
- a substring of a string using `Substring.getc`.

Here is an example of a function that transforms a character stream by splitting the stream into lines and extracting a triple of an integer, boolean and a real from each line.

```
fun ibr rdr the_cstrm =
let
    (*  Read all characters to the end of the
        string or a newline.  This returns the
        line and the rest of the stream.
    *)
    fun get_line cstrm rev_line =
    (
        case rdr cstrm of
          NONE => (cstrm, implode(rev rev_line))
                            (* ran out of chars *)

        | SOME (c, rest) =>
        (
            if c = #"\n"
            then
                (rest, implode(rev rev_line))
            else
                get_line rest (c::rev_line)
        )
    )

    val (strm_out, line) = get_line the_cstrm []
    val l1 = Substring.all line

    val (i, l2) = valOf(Int.scan StringCvt.DEC
                            Substring.getc l1)
    val (b, l3) = valOf(Bool.scan Substring.getc l2)
    val (r, l4) = valOf(Real.scan Substring.getc l3)
in
    SOME((i, b, r), strm_out)
end
```

```
handle Option => NONE
```

The transformer must take a character reader as an argument so I want the expression `(ibr rdr)` to be a reader that reads triples when `rdr` is a character reader. But a reader is a function taking a stream as an argument. So if I define the function as `fun ibr rdr the_cstrm` where `the_cstrm` is the character stream to read then, using currying, the expression `(ibr rdr)` will be of the correct type, i.e. a function taking a character stream.

The first thing the transformer does is read characters from the stream until it has a complete line. The characters are accumulated into a list in reverse. At the end they are joined into a string. This is the fastest way to accumulate a string from characters.

Next I want to get the line into a stream that can be scanned. The `Substring.getc` function satisfies the requirements for a reader function if the stream is a substring. Then I can call the scan functions for each type to get the values on the line. Note that I get back an updated substring in the `l2`, `l3` and `l4` variables. I use `valOf` to get the result out from under the `SOME`. If the scan fails then it will return `NONE` which will cause `valOf` to raise the `Option` exception. A reader function indicates failure by returning `NONE` so that's what the exception handler does.

The main program shows how `StringCvt.scanString` applies the transformer to a string returning exactly one result.

```
fun main(arg0, argv) =
let
    val text = "\
            \  123 true  23.4       \n\
            \  -1  false -1.3e3     \n\
            \"
in
    case StringCvt.scanString ibr text of
      NONE => print "ibr failed\n"

    | SOME (i, b, r) => print(concat[
            Int.toString i, " ",
            Bool.toString b, " ",
            Real.toString r, "\n"
```

```
            ]);

    OS.Process.success
end
```

See the documentation for `StringCvt` for more details.

Bytes

Bytes are represented by the type `word` in the `Word8` structure. The `Byte` structure provides conversions between strings of characters and sequences of bytes. This will be especially useful for the web server project since reading and writing to TCP/IP sockets is done with bytes which we will want to convert to strings.

Integers

The Basis library allows for multiple modules for signed integers of different sizes. These all satisfy the common signature `INTEGER`. On 32 bit architectures SML/NJ only supplies 31 and 32 bit signed integers.

The 31 bit size looks strange but it is designed to help the garbage collector (GC). The GC needs to scan each word in the heap and tell whether it is a pointer or not. The SML/NJ solution is to use the least significant bit (LSB) to tell the difference. If you want a 32 bit integer then it must be allocated in a separate heap record with a type tag. (See the section called *Heap Object Layout* in Chapter 7 for details). You should use 31 bit integers wherever possible for more efficient storage.

The structures `Int` and `Int31` are both 31 bit integers and the top-level type `int` is the same as `Int.int`. Use the `Int32` structure for 32 bit integers. There is also a structure called `LargeInt` which is supposed to be the largest integer type that the hardware implements. It is used as an

intermediary when converting between integers of other sizes. The `bind-largest.sml` file in the compiler's `boot` directory shows the definitions of the various integer types in terms of sizes. (The `Position` type is used for file positions in the Posix functions).

To get an integer literal of a type other than Int.int you will need a type constraint e.g. write `(23: Int32.int)` to get the 32 bit integer value 23. You can use a `0x` prefix for hexadecimal, like in C.

All integer operations check for overflow, unlike C. The `Overflow` exception is raised if there is any integer overflow or underflow.

There are also matching unsigned integer structures, `Word`, `Word8`, `Word31`, `Word32`, `LargeWord` and `SysWord`. These are of sizes 8, 31 and 32 bits as you would expect. Use a `0w` prefix for word values and `0wx` for word values in hexadecimal. Arithmetic on unsigned integers does not raise the Overflow exception.

The unsigned integer structures conform to the `WORD` signature which adds bit-wise operations to the usual integer operations.

Finally there is an infinite precision integer structure `IntInf`. This represents numbers with any number of digits. To input a literal `IntInf.int` value you need to use `fromString` i.e. write

```
valOf(IntInf.fromString "12345678987654321000000000")
```

Words can be serialised into byte arrays using the `PACK_WORD` structures. These implement 16 and 32 bit serialisation in big or little endian order in the `Pack16Big`, `Pack32Big`, `Pack16Little` and `Pack32Little` structures.

Reals

The signature `REAL` describes the floating-point numbers in SML. Just as with the integer types there can be multiple structures for reals of different

precision. But in the SML/NJ compiler there is just the single structure `Real64` and `Real` and `LargeReal` are aliases for it. The top-level type `real` is an alias for `Real.real`.

The design of the `Real` structure aims for correctness first and foremost. This means it supports the IEEE 754 standard on Nan and infinity and signed zero. All operations are checked and exceptions, such as `Div` and `Overflow`, are raised. See the documentation for details.

What is unusual about the `real` type is that it does not allow the standard equality operators, "=" and "<>". This is to avoid the usual confusion of equality with inexact values. Instead the Real structure provides a collection of comparison operators. You can use `Real.==` and `Real.!=` for equality checking with IEEE semantics. These are prefix functions on pairs so you would have to write `(Real.==(x, y))`. The top-level "<", "<=", ">" and ">=" are still available as infix operators on the reals.

The `Real` structure includes a version of a math structure, conforming to the `MATH` signature, that provides trigonometric and logarithmic functions over the real type in `Real`. However these functions are implemented directly in SML rather than using an underlying math library such as the C library. This makes them a bit slow.

The Basis definition describes Pack structures to serialise reals but unfortunately they are not implemented in SML/NJ.

Lists

The `List` structure provides a typical collection of list operations.

The "@" operator will append two lists but this will result in copying the first list in order that it remain immutable. Only prepending an element with "`::`" is cheap. You can prepend in bulk using the `revAppend` function. This will push all of the elements of the first list onto the front of the second. This is useful when building lists in reverse in tail-recursive functions.

The apply function, `app`, applies a function only for side-effects. These might be for writing to an output stream or updating an imperative table.

The tabulate function can be useful for creating lists of a given length. For example

```
implode(List.tabulate(100, fn _ => #" "))
```

will make a string of 100 blanks.

The `ListPair` structure adds some functions for dealing with lists of pairs.

Arrays and Vectors

Arrays are the only other mutable data structure after the `ref` type. They are similar to C arrays. Entries are indexed from 0. All index operations are range checked. The top level `array` type can contain entries of any type (as long as they are all the same type of course). In addition SML/NJ provides arrays specialised to a fixed entry type: `CharArray`, `Word8Array` and `Real64Array`. You could use `CharArray` as a text buffer or `Word8Array` as a byte buffer. These conform to the `MONO_ARRAY` signature.

If you want faster access without range checking then there are some unsafe versions of these structures in `Unsafe.CharArray`, `Unsafe.Word8Array` and `Unsafe.Real64Array`. See the section called *The Unsafe API* in Chapter 4.

The `Array2` structure provides two-dimensional arrays.

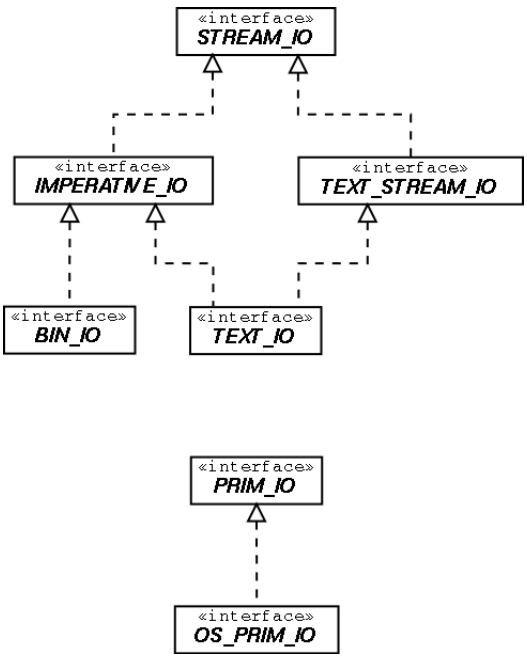
Vector types are like arrays but they are immutable. The most useful forms of vectors are the monomorphic structures: `CharVector`, `Word8Vector` and `Real64Vector`. The `CharVector.vector` type is defined to be the same as the top-level string type so you can use its functions on strings. The `Word8Vector.vector` provides a byte string that is used for binary I/O in other parts of the Basis library.

The Portable I/O API

The Basis library provides a portable I/O API built on top of the operating system facilities. The source for most of the API can be found in the `boot/IO` directory of the compiler. The OS-dependent part of the implementation can be found in the `boot/Unix` directory for Posix-based Unix systems.

Figure 3-2 shows the major interfaces of the I/O API. (The notation is based on UML). SML signatures are pure interfaces and are extended by refinement which adds new features and by specialisation which makes abstract types concrete.

Figure 3-2. The Major Signatures of the Portable I/O API



The lowest level interface is `PRIM_IO`. It abstracts the basic operations of

reading and writing over some I/O channel. The `OS_PRIM_IO` interface extends this with some functions for associating a channel with a file via some sort of OS-dependent file descriptor or handle.

The next level up is the `STREAM_IO` interface. It wraps buffering operations around the I/O channels and calls them streams. It is abstract over any implementation of I/O channels and any type of data element. (The `PRIM_IO` interface will be used to provide an implementation for `STREAM_IO` later).

Input streams are handled in a lazy functional manner. This means that streams are read from only upon demand (as you would expect) and the read returns a stream updated at a new position. So you can read from the same stream value multiple times and it always returns the same data from the same position. Output streams are imperative. Each write will append new data to the output stream.

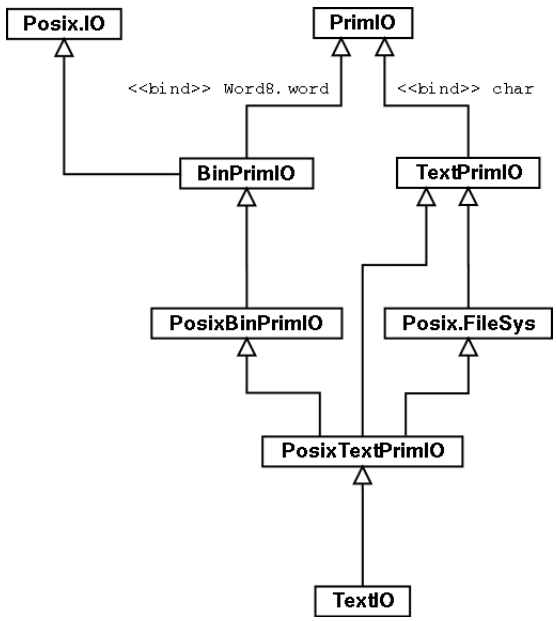
The `TEXT_STREAM_IO` interface specialises `STREAM_IO` for characters and extends it with a function to read a line of text and to write from substrings (see the `Substring` structure).

The `IMPERATIVE_IO` interface wraps around the `STREAM_IO` and provides an interface using imperative streams. This means that the position of a stream is a hidden state variable which is updated after each read operation. The `IMPERATIVE_IO` is then specialised into binary and text I/O. The `BIN_IO` interface fixes the data type to be bytes. The `TEXT_IO` fixes it to be characters with an understanding of text conventions such as line splitting. It also gets knowledge of the Unix stdin, stdout and stderr text streams. The underlying `STREAM_IO` interface is made visible in `TEXT_IO` for when you want to use a functional I/O style.

The implementation hierarchy is shown in Figure 3-3. The `PrimIO` structure mainly defines some types and utility routines which represent buffered I/O. These are specialised by having the I/O types bound to either bytes (`Word8.word`) or characters. This results in the structures `BinPrimIO` and `TextPrimIO` respectively. The `PosixBinPrimIO` structure adds an implementation of binary I/O using the I/O functions in `Posix.IO`. (See the section called *`Posix.IO`*). Then the `PosixTextPrimIO` structure casts the

binary I/O to text I/O and this results in the `TextIO` structure.

Figure 3-3. The Major Structures Implementing the Portable I/O API



For binary I/O there is a matching `BinIO` structure that reads and writes streams of bytes.

There is an `IO` structure that defines some common exceptions and types. The `IO.Io` exception is the main error reporting mechanism for the portable I/O API.

Here is a simple example that counts characters, words and lines in a file. It does the job of the Unix `wc` command but without the command line options. Here is the main function.

```
fun main(arg0, argv) =
let
in
```

```
    case argv of
      [] => count TextIO.stdIn ""

    | (file::_) =>
        let
            val strm = TextIO.openIn file
        in
            (count strm file) handle x =>
                (TextIO.closeIn strm; raise x);
            TextIO.closeIn strm
        end;

    OS.Process.success
end
handle
  IO.Io {name, function, cause} =>
    (
        toErr(concat["IO Error: ", name,
                     ", ", exnMessage cause, "\n"]);
        OS.Process.failure
    )

| x => (toErr(concat["Uncaught exception: ", exnMessage x,"\n"]);
        OS.Process.failure)
```

If there are no command line arguments then I read from stdin. If there are some then I take the first one and ignore the rest. Any I/O exception from the count function for a file is caught so that we can close the file. This is not strictly necessary since the file will get closed anyway when the program exits but I included it as an example of catching and reraising an exception. An I/O exception from anywhere else will be caught in the outermost handlers down the bottom.

Here is the `count` function. It is just a simple loop which terminates when the `inputLine` function returns an empty string. An empty line does not terminate the loop since it will have a new-line character in it. The `inputLine` function also returns a new-line in the case of an unterminated last line in a file. So the program will count an extra character in this case. Words are counted by splitting the line into tokens at white space and counting how many we get.

```
fun count strm file =
```

```
let
    fun read (nchars, nwords, nlines) =
    (
        (* This ensures the line ends with a \n
           unless we are at eof.
        *)
        case TextIO.inputLine strm of
          "" => (nchars, nwords, nlines)

        | line =>
            let
                val words = String.tokens Char.isSpace line
            in
                read (nchars + size line,
                      nwords + length words,
                      nlines + 1)
            end
    )

    val (nchars, nwords, nlines) = read (0, 0, 0)
in
    print(concat[Int.toString nlines, " ",
                 Int.toString nwords, " ",
                 Int.toString nchars, " ",
                 file, "\n"])
end
```

The Portable OS API

The Basis library provides a portable API for operating system-level operations. These deal in processes, file systems, time, date and device-specific I/O attributes, and models for manipulating these resources that are largely system-independent. In the Basis documentation this API is described under the heading of "System".

The `OS` structure contains part of the API in the substructures `FileSys`, `Path`, `Process` and `IO` as well as some common exception and error handling functions. The `OS.FileSys` structure provides an API for scanning directories, altering directories by deleting or renaming files and checking access permission. You can get and change the current directory here too.

The `OS.Path` provides an abstract view of file paths. The `OS.Process` structure provides a few process-oriented functions that are still rather Unix specific. The `OS.IO` structure provides an interface to the Posix `poll` system call. Presumably it can be implemented in terms of other system calls on other operating systems. I'll ignore it.

The signatures for the OS API can be found in the `boot/OS` directory of the compiler. Most of the implementation can be found in the `boot/Unix` directory for Posix-based Unix systems.

`OS.FileSys`

The file system API is straight-forward enough. I'll illustrate it with a program to scan a directory tree looking for writable executable files on the grounds that these might be a security hazard. Symbolic links won't be followed. I start with some useful declarations.

```
structure FS = OS.FileSys
structure OP = OS.Path

fun toErr msg = TextIO.output(TextIO.stdErr, msg)

exception Error of string
```

The structure declarations just provide convenient shorthand names. I've added my own Error exception so that I can report more meaningful context-sensitive errors.

Here is the main function.

```
fun main(arg0, argv) =
let
in
    case argv of
      [] => scan_dir OP.currentArc

    | (file::_) => scan_dir file;

    OS.Process.success
end
```

```
handle
  OS.SysErr (msg, _) =>
    (
        toErr(concat["System Error: ", msg, "\n"]);
        OS.Process.failure
    )

| Error msg => (toErr msg; OS.Process.failure)

| x => (toErr(concat["Uncaught exception: ", exnMessage x,"\n"]);
        OS.Process.failure)
```

The program will take a directory name on the command line or if one is omitted then the current directory is used. The `OS.Path.currentArc` function provides a portable way to represent the current directory. Errors that I expect from user input are reported via the `Error` handler. But I also catch other `OS.SysErr` exceptions just in case.

I open a directory with the following function. It catches the `OS.SysErr` exception and turns it into a more meaningful error message. (The optional `syserror` code in the exception is redundant).

```
fun open_dir dir =
(
    FS.openDir dir
)
handle
  OS.SysErr (msg, _) => raise Error (concat[
            "Cannot open directory ", dir, ": ", msg, "\n"])
```

Finally here is the directory scanning function.

```
fun scan_dir dir =
let
    (* val _ = print(concat["scan_dir ", dir, "\n"]) *)
    val strm = open_dir dir

    fun get_files files =
    (
        case FS.readDir strm of
          "" => rev files           (* done *)

        | f =>
```

```
            let
                val file = OP.joinDirFile
                            {dir = dir, file = f}
            in
                if FS.isLink file
                then
                    get_files files
                else
                    get_files (file::files)
            end
    )

    val files = get_files []
    val _     = FS.closeDir strm

    fun show_wx file =
    (
        if FS.access(file, [FS.A_WRITE, FS.A_EXEC])
        then
            (print file; print "\n")
        else
            ()
    )

    fun scan_subdir file =
    (
        if FS.isDir file
        then
            scan_dir file
        else
            ()
    )
in
    app show_wx files;
    app scan_subdir files
end
```

The first line is just some tracing I used while debugging the function.

The `val` declarations are executed in the order that they appear. This is important since they may have side-effects. The `get_files` function reads the stream to build a list of files in the directory. It comes after the `strm` name is defined because it refers to it as a global name. The directory stream is updated imperatively by the `readDir` function. The

`OS.Path.joinDirFile` function is a portable way to add a path element. It will use the right kind of slash.

I want to avoid accumulating open directories while walking the directory tree as I might run out of open files if the tree is too deep. So instead I extract the files and close the directory stream. This means that I will be retaining in memory all of the files in the directories along a path through the tree. File paths will be discarded on the way back up the tree so the garbage collector can reclaim them if needed. There is lots more memory than there are available file descriptors.

The `show_wx` function prints the file name if it is writable and executable. It is iterated over the list of files using the built-in `app` function (see the section called *Lists*). I recurse in `scan_subdir` by scanning each file that is a directory.

`OS.Path`

The functions in `OS.Path` model a file path as a list of names called arcs. There is also provision for a volume name for Microsoft Windows. File names can have an extension marked by a "." character. There are functions for splitting and joining all of these kinds of parts.

`OS.Process`

Your main interest in `OS.Process` is for the `success` and `failure` values which are needed as exit codes for your main program. The `system` command for running shell commands could be useful but if you want to capture the output see the functions in the `Unix` structure.

You can abort your program early with the `exit` or `terminate` functions but I prefer to use an exception for fatal errors. It leaves open the possibility of higher-level functions trapping the errors.

The `getEnv` function gets environment variables like the C library's `getenv()` function.

Time and Date

Time and Date values are provided in the `Time` and `Date` structures respectively. They are documented in the Basis library under the "System" heading. Their implementations appear in the `boot/` directory of the compiler.

Time values in SML/NJ are stored as a pair of seconds and microseconds.

The main trick to remember with the `time` type is that the top-level operators are not overloaded on them. So if you want to subtract two time values you need to write something like `Time.-(t1, t2)`. Similarly for the other symbolic functions in `Time`.

Also the conversion with integer values uses the `LargeInt` structure which on 32-bit systems is the same as `Int32` i.e. boxed 32-bit integers. (See the section called *Integers*).

You can convert a date represented as the number of seconds since the Unix epoch to a year/month/day representation using the Time and Date structures. Write something like this:

```
val date = Date.fromTimeLocal(Time.fromSeconds 99999999)
val year = Date.year date
```

The month and day of the week is represented by the enumeration types `weekday` and `month` in `Date`. There is no mechanism for converting these values to integers or strings. You'll have to write your own. What you can do though is use the `Date.fmt` function to format a date any way you like. It uses the `strftime()` C library function underneath so the result should be properly locale-dependent.

At the time of writing, SML/NJ has not implemented the `Date.fromString` and `Date.scan` functions.

Unix

The "System" documention for the Basis library describes a Unix structure. It's not really a portable OS thing. It just provides some simple utility functions for spawning a sub-process, talking to it through stdin and stdout and killing it afterwards. The source for it can be found in the `boot/Unix` directory of the compiler.

The POSIX API

The Basis library provides a useful collection of POSIX functions on Unix systems. These are grouped together under substructures in the `Posix` structure. The source for these functions can be found in the compiler in the `boot/Posix` directory surprisingly enough. Much of the SML implementation is just a wrapper around calls to the corresponding C functions. The C code is in the runtime under the `c-libs/` directory.

`Posix.Error`

The `syserror` value is a representation of the POSIX `errno` error codes. This is the same type as `OS.syserror` which appears in the `OS.SysErr` exception.

It appears as an abstract type but internally it is represented as an integer with the same value as the `errno` code. The `errorMsg` function returns the same string used by the `perror()` C library function. The unique name returned by `errorName` is derived from the symbol for the POSIX error code. If you need the actual integer value for the error code then you can use `toWord`.

Posix.FileSys

The Posix.FileSys structure provides functions for dealing with directories and files except for the actual I/O which is in Posix.IO. Where possible you should use the corresponding OS.FileSys functions which are intended to be a bit more portable.

At this level you are working with Unix file descriptors, represented by the type file_desc. There is also a OS.IO.iodesc type for the file descriptors used by the poll interface in OS.IO. A separate type is used to make the OS.IO interface more independent of the POSIX layer and therefore more portable. Underneath they are both Unix file descriptors.

Most of the functions should be straight-forward to use. The flags may need some explaining. Flags are represented by lists of values of some flag type. The underlying bit patterns of the values in a list are or-ed together. Each substructure containing flags inherits a flags function from the POSIX_FLAGS signature to convert the list to the bit pattern. For example write

```
structure FS = Posix.FileSys
...
    FS.chmod("myfile", FS.S.flags
                [FS.S.irusr, FS.S.irgrp, FS.S.iroth])
...
```

to set the permissions of the file "myfile" to 0444. The sticky mode bit with value 01000 is deliberately filtered out by the stat functions as it is not part of the POSIX standard so you will never be able to test or preserve it.

To give the functions a workout here is a stat (basis/statx.sml) program that reports a file's status in detail. First I start with some useful declarations. The wordToDec function is needed since the SysWord.toString function always formats in hexadecimal. (See the Basis documentation on the WORD signature).

```
structure FS = Posix.FileSys

exception Error of string
```

```
fun toErr msg = TextIO.output(TextIO.stdErr, msg)

fun wordToDec w = SysWord.fmt StringCvt.DEC w
```

Here is the main function. It is pretty boiler-plate by now. It only recognises a single command line argument which must be the file name. The various functions we are using use the `OS.SysErr` exception so I've put in a catch-all for it.

```
fun main(arg0, argv) =
let
in
    case argv of
      [file] => (stat file; OS.Process.success)

    | _ => (toErr "Usage: statx filename\n"; OS.Process.failure)
end
handle
  OS.SysErr (msg, _) =>
    (
        toErr(concat["Error: ", msg, "\n"]);
        OS.Process.failure
    )

| Error msg => (toErr msg; OS.Process.failure)

| x => (toErr(concat["Uncaught exception: ", exnMessage x,"\n"]);
        OS.Process.failure)
```

Here is the function to report the stat data. I've put in a `SysErr` handler on the `stat` function so that I can report the file name. This is the most likely error to come from the program. Note that for better portability you should use the matching integer structure when printing integers e.g. `Position.toString` for `Position.int` types. It happens on the Intel x86 architecture that `Position` = `Int` but this may not be true on other architectures.

```
fun stat file =
let
    val st = (FS.stat file)
                handle OS.SysErr (msg, _) =>
                    raise Error (concat[ file, ": ", msg, "\n"])
```

```
    val mode = FS.ST.mode st
    val uid  = FS.ST.uid st
    val gid  = FS.ST.gid st
in
    print(concat["  File: ", file, "\n"]);

    print(concat["  Size: ",
        Position.toString(FS.ST.size st), "\n"]);

    print(concat["  Type: ",
        filetypeToString st, "\n"]);

    print(concat["  Mode: ",
        SysWord.fmt StringCvt.OCT (FS.S.toWord mode),
        "/", modeToString mode, "\n"]);

    print(concat["   Uid: ",
        uidToInt uid, "/", uidToName uid, "\n"]);

    print(concat["   Gid: ",
        gidToInt gid, "/", gidToName gid, "\n"]);

    print(concat["Device: ",
        devToString(FS.ST.dev st), "\n"]);

    print(concat[" Inode: ",
        wordToDec(FS.inoToWord(FS.ST.ino st)), "\n"]);

    print(concat[" Links: ",
        Int.toString(FS.ST.nlink st), "\n"]);

    print(concat["Access: ", Date.toString(
        Date.fromTimeLocal(FS.ST.atime st)), "\n"]);

    print(concat["Modify: ", Date.toString(
        Date.fromTimeLocal(FS.ST.mtime st)), "\n"]);

    print(concat["Change: ", Date.toString(
        Date.fromTimeLocal(FS.ST.ctime st)), "\n"]);
    ()
end
```

The first of the helper functions called from `stat` is `filetypeToString`. It searches a list of predicate functions to find one that works on the stat buffer

value. The matching name is returned. I've put the list of predicates within a *local* block so that is private to `filetypeToString` without being inside it. This way the list isn't built every time that the function is called, which is wasteful. This doesn't matter on this small program but it very well might in other programs.

```
local
    val type_preds = [
            (FS.ST.isDir,   "Directory"),
            (FS.ST.isChr,   "Char Device"),
            (FS.ST.isBlk,   "Block Device"),
            (FS.ST.isReg,   "Regular File"),
            (FS.ST.isFIFO,  "FIFO"),
            (FS.ST.isLink,  "Symbolic Link"),
            (FS.ST.isSock,  "Socket")
            ]
in
    fun filetypeToString st =
    let
        val pred = List.find (fn (pred, _) => pred st) type_preds
    in
        case pred of
          SOME (_, name) => name
        | NONE => "Unknown"
    end
end
```

The `modeToString` helper function iterates over a list of flag testing functions, one for each position in the final mode string. I've used currying to make each of the expressions in the list a function from a mode to the character for the mode in the string.

```
local
    fun test flag ch mode =
    (
        if FS.S.anySet(FS.S.flags [flag], mode)
        then
            ch
        else
            #"-"
    )

    fun test2 flag1 ch1 flag2 ch2 mode =
```

```
    (
        if FS.S.anySet(FS.S.flags [flag1], mode)
        then
            ch1
        else
            if FS.S.anySet(FS.S.flags [flag2], mode)
            then
                ch2
            else
                #"-"
    )

    val flags  = [
            test  FS.S.irusr #"r",
            test  FS.S.iwusr #"w",
            test2 FS.S.isuid #"s" FS.S.ixusr #"x",
            test  FS.S.irgrp #"r",
            test  FS.S.iwgrp #"w",
            test2 FS.S.isgid #"s" FS.S.ixusr #"x",
            test  FS.S.iroth #"r",
            test  FS.S.iwoth #"w",
            test  FS.S.ixoth #"x"
            ]
in
    fun modeToString mode =
    let
        val chars = foldl
            (fn (func, rslt) => (func mode)::rslt)
            [] flags
    in
        implode(rev chars)
    end
end
```

The next group of functions convert uids and gids to their string forms, both as a decimal number and as a name from the passwd/group files. These use functions from the `Posix.ProcEnv` and `Posix.SysDB` structures, described later. If there is any exception then I assume that the name is not known in the file.

```
local
    structure PROC = Posix.ProcEnv
    structure DB   = Posix.SysDB
in
    fun uidToInt uid = wordToDec(PROC.uidToWord uid)
```

```
    pun gidToInt gid = wordToDec(PROC.gidToWord gid)

    fun uidToName uid =
    (
        (DB.Passwd.name(DB.getpwuid uid))
            handle _ => "unknown"
    )

    fun gidToName gid =
    (
        (DB.Group.name(DB.getgrgid gid))
            handle _ => "unknown"
    )
end
```

Finally here is `devToString`. I need to do some bit operations to separate the bytes of the `dev_t` word. The current SML definition for a device value does not allow for the newer 64-bit device numbers. But on Linux on Intel x86 I get the major and minor numbers in the lower 16 bits of the word. This is not very portable.

```
fun devToString dev =
let
    val word = FS.devToWord dev
    val w1 = SysWord.andb(SysWord.»(word, 0w8), 0wxff)
    val w2 = SysWord.andb(word, 0wxff)
in
    concat[wordToDec w1, ",", wordToDec w2]
end
```

POSIX_FLAGS

This signature is an interface that is inherited into each distinct set of flags in other `Posix` structures. See for example `Posix.FileSys.S` for the mode flags. It provides common operations on flags which are represented as bit-strings internally. See the section called *`Posix.FileSys`* for an example of flag use.

Posix.IO

This structure provides the functions that deal with the content of files as a stream of bytes. It works with the file descriptors that were created with the `Posix.FileSys` functions. There is not a lot of need to use the read and write functions in this structure for general purpose binary file I/O as the `BinIO` structure in the section called *The Portable I/O API* should be all that you will need. You could use them in conjunction with other functions that deal with file descriptors such as the file locking functions.

A good demonstration of programming at this level can be found in the implementation of the `execute` function in the `Unix` structure. (See the `boot/Unix` directory of the compiler). It shows how to fork and exec a child process and build portable I/O streams from file descriptors. Central to building I/O streams are the `mkReader` and `mkWriter` functions that are declared in the `OS_PRIM_IO` signature. (See the section called *The Portable I/O API*). These build reader and writer objects for buffered I/O given a POSIX file descriptor. You can find implementations of them in the `PosixBinPrimIO` and `PosixTextPrimIO` structures. The result is this code from the `Unix` structure.

```
fun fdReader (name : string, fd : PIO.file_desc) =
      PosixTextPrimIO.mkReader {
          initBlkMode = true,
          name = name,
          fd = fd
        }

fun fdWriter (name, fd) =
      PosixTextPrimIO.mkWriter {
          appendMode = false,
          initBlkMode = true,
          name = name,
          chunkSize=4096,
          fd = fd
        }

fun openOutFD (name, fd) =
      TextIO.mkOutstream (
        TextIO.StreamIO.mkOutstream (
          fdWriter (name, fd), IO.BLOCK_BUF))
```

```
fun openInFD (name, fd) =
      TextIO.mkInstream (
        TextIO.StreamIO.mkInstream (
          fdReader (name, fd), NONE))
```

The `name` argument is only used for error reporting to distinguish the stream. The implementation in the `PosixBinPrimIO` and `PosixTextPrimIO` structures use the `Posix.IO.setfl` function to change the blocking mode as requested by the blocking and non-blocking versions of the I/O functions in a reader or writer. You need to supply the correct initial state for these modes. If you opened the file with, for example, `Posix.FileSys.openf` with `O_APPEND` or `O_NONBLOCK` using the flags in `Posix.FileSys.O` then you must pass in the appropriate values for `initBlkMode` and `appendMode`.

The `openOutFD` and `openInFD` functions assemble the stream layers as shown in Figure 3-2. The output stream is set to be fully buffered. Other possible buffered modes, from the `IO` structure, are `NO_BUF` for no buffering at all and `LINE_BUF` if you want to flush the buffer after each line of text. (`LINE_BUF` is the same as `BLOCK_BUF` for binary streams).

Once you have built a stream on a file descriptor you cannot easily retrieve the file descriptor to manipulate it while the stream is live. If you call `TextIO.StreamIO.getReader` for example intending to get the reader's `ioDesc` field then the stream will be terminated on the assumption that you will be taking over all I/O from then on. If you need access to the file descriptor then you should save it somewhere yourself. You might do this if you want to use the polling interface of the `OS.IO` structure. (The `canInput` function on streams doesn't poll, it just attempts to do a non-blocking read on the file descriptor).

Here is the code for `Unix.executeInEnv`. It demonstrates file descriptor manipulation while forking and setting up some pipes.

```
structure P = Posix.Process
structure PIO = Posix.IO
structure SS = Substring
```

```
fun executeInEnv (cmd, argv, env) = let
      val p1 = PIO.pipe ()
      val p2 = PIO.pipe ()

      fun closep () = (
            PIO.close (#outfd p1);
            PIO.close (#infd p1);
            PIO.close (#outfd p2);
            PIO.close (#infd p2)
          )

      val base = SS.string(SS.taker
                  (fn c => c <> #"/") (SS.all cmd))

      fun startChild () =
      (
        case protect P.fork () of
          SOME pid =>  pid                      (* parent *)

        | NONE =>
        let
            val oldin = #infd p1
            val newin = Posix.FileSys.wordToFD 0w0

            val oldout = #outfd p2
            val newout = Posix.FileSys.wordToFD 0w1
        in
            PIO.close (#outfd p1);
            PIO.close (#infd p2);

            if (oldin = newin) then () else (
                PIO.dup2{old = oldin, new = newin};
                PIO.close oldin);

            if (oldout = newout) then () else (
                PIO.dup2{old = oldout, new = newout};
                PIO.close oldout);

            P.exece (cmd, base::argv, env)
        end

      val _ = TextIO.flushOut TextIO.stdOut

      val pid = (startChild ())
                  handle ex => (closep(); raise ex)
```

```
    val ins = openInFD (base^"_exec_in", #infd p2)
    val outs = openOutFD (base^"_exec_out", #outfd p1)

in
  (* close the child-side fds *)
  PIO.close (#outfd p2);
  PIO.close (#infd p1);

  (* set the fds close on exec *)
  PIO.setfd (#infd p2, PIO.FD.flags [PIO.FD.cloexec]);
  PIO.setfd (#outfd p1,PIO.FD.flags [PIO.FD.cloexec]);

  PROC {
    pid = pid,
    ins = ins,
    outs = outs
  }
end
```

The `startChild` function forks (see the section called *`Posix.Process` and `Posix.Signal`*) and dups file descriptors in the usual way to get the pipes connected to stdin and stdout while being careful that they are not already connected that way. Remember to close the unused ends of the pipe in the parent and child or else you won't be able to get an end-of-file indication when the child exits.

Posix.ProcEnv

This structure provides access to information about a process such as its uid, gid, pid, running time or environment variables.

You can also get system information via the `uname` and `sysconf` functions. You form the string argument to `sysconf` by deleting the _SC_ prefix from the POSIX value name, for example to get _SC_OPEN_MAX write `Posix.ProcEnv.sysconf "OPEN_MAX"`. All of the _SC_ values defined in `unistd.h` on your system should be available this way.

To use file descriptors with `isatty` you need the conversion function in `Posix.FileSys`. For example to determine if stdin is a tty:

```
fun isatty() = Posix.ProcEnv.isatty (Posix.FileSys.wordToFD 0w0)
```

Posix.Process and Posix.Signal

This structure provides functions to fork and exec processes, kill and wait for them. Equivalent functions for the C library's `alarm()`, `pause()` and `sleep()` functions are also included. You can find a demonstration of fork and exec in the section called *`Posix.IO`*.

The `kill` function uses the signal values defined in `Posix.Signal`. This defines a type `signal` with values for each of the POSIX signals. You can also convert these to the integer codes for your platform with the `toWord` function.

Unfortunately the POSIX API does not currently provide for setting signal handlers. For that you need to resort the older signal API of the SMLofNJ structure in the section called *Signals* in Chapter 4. (If you are looking in the `boot/Unix` directory of the compiler, the `unix-signals*` files define the signals for this older API).

Posix.SysDB

This structure provides an API for reading the `/etc/passwd` and `/etc/group` files. The `uidToName` function in the `statx` program of the section called *`Posix.FileSys`* provides a little demonstration of the API.

Posix.TTY

This structure provides a termio-style API to terminals.

The following function from the `ttyx` (ttyx.sml) program shows how to change the erase character on your terminal. (Updating a single field in a

record is a pain in SML).

```
fun setErase ch =
let
    val fd   = Posix.FileSys.wordToFD 0w0
    val attr = TTY.getattr fd
    val new_attr = TTY.termios {
            iflag  = TTY.getiflag attr,
            oflag  = TTY.getoflag attr,
            cflag  = TTY.getcflag attr,
            lflag  = TTY.getlflag attr,
            cc     = TTY.V.update
                        (TTY.getcc attr, [(TTY.V.erase, ch)]),
            ispeed = TTY.getispeed attr,
            ospeed = TTY.getospeed attr
            }
in
    TTY.setattr(fd, TTY.TC.sanow, new_attr)
end
```

Note that at the time of writing this, the Basis library documentation for `Posix.TTY` doesn't match SML/NJ version 110.0.7. In version 110.0.7 there is no internal structure called `Posix.TTY.CF`. Its contents appear directly in `Posix.TTY`. Similarly these functions which should be in the `Posix.TTY.TC` structure appear directly in `Posix.TTY`: `getattr`, `setattr`, `sendbreak`, `drain`, `flush`, and `flow`.

Chapter 4. The SML/NJ Extensions

These are extensions to the Basis library that are specific to SML/NJ. You can find reference documentation to them in the "Special features of SML/NJ" page via the SML/NJ home page[*SML*].

The Unsafe API

The Unsafe API is a collection of functions that bypass the normal safety checks of the language and the Basis library. These functions are available in the `Unsafe` structure. It provides:

- Access to the elements of arrays and vectors, including strings, without the usual subscript range checks.
- Access to information about the memory representation of values.
- An interface to C functions in the runtime.
- Miscellaneous operations used internally by the compiler and associated subsystems.

Unchecked subscripting is used internally by the array and vector functions in the Basis library. Wherever possible you should design your code to make use of the Basis functions. Using the unchecked operations directly puts your program at risk of crashing.

Unsafe Vectors and Arrays

The following monomorphic vectors and arrays are available.

`Unsafe.CharVector`

This operates on strings, which are vectors of characters.

`Unsafe.Word8Vector`

This operates on vectors of bytes.

`Unsafe.CharArray`
`Unsafe.Word8Array`

These operate on arrays of characters or bytes.

`Unsafe.Real64Array`

This operates on arrays of double precision reals. The C equivalent would be the array type `double[]`.[1]

These structures conform to one of these two signatures.

```
signature UNSAFE_MONO_VECTOR =
  sig

    type vector
    type elem

    val sub : (vector * int) -> elem
    val update : (vector * int * elem) -> unit
    val create : int -> vector

  end

signature UNSAFE_MONO_ARRAY =
  sig

    type array
    type elem

    val sub : (array * int) -> elem
    val update : (array * int * elem) -> unit
    val create : int -> array

  end
```

So you can see that you get to update elements of vectors in place just as you can with arrays. The `create` functions create a vector or array of the given length with uninitialised elements.

For arrays and vectors of other kinds of elements there are the structures `Unsafe.Vector` and `Unsafe.Array` which conform to the following signatures.

```
signature UNSAFE_VECTOR =
  sig

    val sub : ('a vector * int) -> 'a
    val create : (int * 'a list) -> 'a vector

  end

signature UNSAFE_ARRAY =
  sig

    val sub : ('a array * int) -> 'a
    val update : ('a array * int * 'a) -> unit
    val create : (int * 'a) -> 'a array

  end
```

The vector `create` function creates a vector from a list. You have to supply the length of the list as the first argument. The array `create` function creates an array given a length and an initial value for each element.

Memory Representation Information

The `Unsafe.Object` structure provides some functions for getting information about the memory representation. Read the source code in the `boot/Unsafe/object*` files of the compiler. You won't find much use for this in your programs. The most useful functions look like being the `toWord32`, `toInt32` functions which can convert a byte array to a 32 bit integer. But there isn't enough functionality here to be useful for serialising

values into a wire protocol. (See the section called *Integers* in Chapter 3 for serialising integers).

You could use this structure to estimate the size of objects in memory. Here is my version of a function to estimate the size of a value, including pointed-to values. I've used `O` as an alias for `Unsafe.Object`.

```
(*  Estimate the size of v in 32-bit words.
    Boxed objects have an extra descriptor word
    which also contains the length for vectors
    and arrays.
*)
fun sizeof v =
let
    fun obj_size obj =
    (
        case O.rep obj of
          O.Unboxed => 1    (* inline 31 bits *)
        | O.Real    => 1+2

        | O.Pair      => tup_size obj
        | O.Record    => tup_size obj
        | O.RealArray => tup_size obj

        | O.PolyArray => arr_size obj

        (* includes Word8Vector.vector
           and CharVector.vector
        *)
        | O.ByteVector => 1 +
            ((size(O.toString obj)+3) div 4)

        (* includes Word8Array.array
           and CharArray.array
        *)
        | O.ByteArray =>  1 +
            ((Array.length(O.toArray obj)+3) div 4)

        | _ => 2    (* punt for other objects *)
    )

    (*  Count the record plus the size of
        pointed-to objects in the heap.
    *)
    and tup_size obj =
    let
```

```
        fun sz obj =
            if O.boxed obj
            then
                1 + (obj_size obj)
            else
                1
    in
        Vector.foldl
            (fn (obj, s) => s + (sz obj))
            1
            (O.toTuple obj)
    end

    and arr_size obj =
    let
        fun sz obj =
            if O.boxed obj
            then
                1 + (obj_size obj)
            else
                1
    in
        Array.foldl
            (fn (obj, s) => s + (sz obj))
            1
            (O.toArray obj)
    end
in
    obj_size(O.toObject v)
end
```

This is a main function to try it out.

```
fun main(arg0, argv) =
let
    fun show name v = print(concat[
            "Size of ", name,
            " = ", Int.toString(sizeof v),
            " 32-bit words\n"])
in
    show "integer"  3;
    show "real"     3.3;
    show "string"   "abc";

    show "pair"     ("abc", 42);
    show "record"   {a = 1, b = 4.5, c = "fred"};
```

```
    OS.Process.success
end
```

See the section called *Heap Object Layout* in Chapter 7 for more information on object layout in the heap.

The C Interface

The runtime includes a collection of C functions that implement the low-level Basis operations such as those in the `Posix` structure. The SML code calls these C functions using the functions in the `Unsafe.CInterface` structure. These functions must be specially written to take arguments in the form of SML values. This is not a general purpose interface to C functions. I only mention it in case you think that it is for general purpose use.

Later versions of SML/NJ will include a general purpose interface for calling any C function in a shared library which is loaded at run-time.

Miscellaneous Unsafe Operations

The `Unsafe.blastRead` and `Unsafe.blastWrite` functions are used to serialise/deserialise entire data structures for writing to files. The `blastWrite` function is expensive to run since it uses the garbage collector to traverse the data structure to locate all values reachable from the root value. You shouldn't call it often to serialise small data structures. Instead it is intended that you build up an entire data structure and then dump it into a file at exit time.

The `Unsafe.cast` function can be used to cast a value to any other type. This of course is very dangerous unless you know the underlying memory representation. Most cases where you might want to do this are already provided for. For example converting between bytes and characters is provided in the `Byte` structure.

The other functions in `Unsafe` should not be used. Some are used by separate systems such as the Concurrent ML library which we will be using later.

The `Unsafe.Poll` structure is not normally accessible and isn't interesting to us.

Signals

The `Signals` structure provides a basic interface to the Unix signal system. It defines four basic signals that can be simulated on most operating systems.

`sigINT`

: On Unix, this corresponds to the SIGINT signal.

`sigALRM`

: This is used by the interval timer in the `IntervalTimer` structure. You can use this to generate periodic interrupts in your program.

`sigTERM`

: On Unix, this corresponds to the SIGTERM signal.

`sigGC`

: This is a pseudo-signal generated internally when a garbage collection has been completed.

In addition the `UnixSignal` structure provides a few more signals found on Unix systems. These are `sigPIPE`, `sigQUIT`, `sigUSR1`, `sigUSR2`, `sigCHLD`, `sigCONT`, `sigTSTP`, `sigTTIN` and `sigTTOU`. See the source code in the `boot/Unix/unix-signals*.sml` files for more details.

There may be more signals available on your platform than appear in `UnixSignal`. In particular `SIGHUP` has missed out on appearing in either structure. You can fetch it yourself like this:

```
    val sighup = valOf(Signals.fromString "HUP")
```

A complete list of the available signals can be printed with this code which you can type to the top-level SML prompt.

```
app (fn s => print(concat[Signals.toString s, "\n"]))
    (Signals.listSignals());
```

Here is a program to demonstrate setting up a simple interrupt handler. It just prints a message and lets the program, which is an infinite loop, continue.

```
fun int_handler(signal, n, cont) =
let
in
    print "interrupt\n";
    cont
end

fun main(arg0, argv) =
let
    fun loop() = (Signals.pause(); loop())
in
    Signals.setHandler(Signals.sigINT,
                       Signals.HANDLER int_handler);
    loop();

    OS.Process.success
end
```

The handler function must return a continuation. This will normally be the third argument to the function. Continuations are described in more detail in the section called *Continuations* in Chapter 6.

A signal handler function does not execute in the same kind of interrupted state as a signal handler in C. The C-level handler queues the signal and completes. A little later the SML-level handler is called as part of the normal

execution of the program. So there are no restrictions on what you can do in an SML signal handler. Also by manipulating the returned continuation you can perform the equivalent of a C `longjmp` to anywhere else in the program.

The `SMLofNJ` API

The `SMLofNJ` structure is a miscellaneous collection of non-standard functions supplied with the SML/NJ compiler. In version 110.0.7 it provides:

- the manipulation of continuations with a Scheme-like call/cc API.
- an interval timer to deliver a periodic trigger to an application.
- a little control over the garbage collector.
- execution time profiling.
- some operating system information.
- utilities for lazy evaluation of a function.
- weak pointers for the garbage collector.
- exporting the heap. This has already been discussed in the section called *Assembling the Hello World Program* in Chapter 2.
- access to an exception history list for debugging.

Call/cc

The call/cc API is in the `SMLofNJ.Cont` structure. It is described in more detail in the section called *Continuations* in Chapter 6.

The Interval Timer

You can set the interval timer to produce alarm signals (`Signals.sigALRM`) at periodic intervals. This can be used to trigger activity in your application. The Concurrent ML library uses it to trigger pre-emptive scheduling of threads so you won't be able to use the interval timer if you are using Concurrent ML. But if you aren't you could write something like the following program.

```
fun alarm_handler(signal, n, cont) =
let
in
    print(concat["tick at ", Time.toString(Time.now()), "\n"]);
    cont
end

fun main(arg0, argv) =
let
    fun loop() = (Signals.pause(); loop())
in
    Signals.setHandler(
        Signals.sigALRM,
        Signals.HANDLER alarm_handler);

    SMLofNJ.IntervalTimer.setIntTimer
        (SOME(Time.fromSeconds 1));

    loop();

    OS.Process.success
end
```

By returning a different continuation you can have your program switch to different code on each clock tick.

Garbage Collection Control

The `SMLofNJ.Internals.GC` provides two functions for control of the garbage collection. Calling `SMLofNJ.Internals.GC.doGC n` with `n` = 0 will trigger a minor collection. With a large value of `n`, say 10, it will trigger

a major collection of all of the generations to reduce the memory usage to a minimum.

You can also turn on or off the collection tracing messages. These are off by default in your programs. Calling `SMLofNJ.Internals.GC.messages true` will turn them on. You will see messages looking like

```
GC #0.1.2.5.6.43:   (60 ms)
GC #1.2.3.6.7.66:   (60 ms)
```

A message is produced for each major collection. The numbers show the number of collections that have been performed in each generation. The oldest generation is on the left. The right-most number is the number of minor collections. The time is the duration of the major collection. The messages can give you some idea of the amount of memory activity in your program and the typical pause times during the collections.

There is more discussion on the SML/NJ implementation of garbage collection in the section called *Garbage Collection Basics* in Chapter 7.

Execution Time Profiling

You access execution time profiling through the `Compiler.Profile` structure, which is separate from the `SMLofNJ` structure. However the profiling uses the low-level control functions in `SMLofNJ.Internals.ProfControl`.

To get profiling you have to compile your program for profiling. Then when it runs it must explicitly turn on the profiling. To compile with profiling using the Compilation Manager you need an extra command. For the example profile program:

```
> CM_ROOT=profile.cm sml
Standard ML of New Jersey, Version 110.0.7, September 28, 2000
- Compiler.Profile.setProfMode true;
- CM.make();
```

It is a good idea to force the recompilation of all of your source when you do this. A simple way to do this is to delete the `CM` directories in each of the source directories of your program. This deletes the cached `.bin` files and they have to be recompiled.

Here is the profile program which just sorts a large list.

```
fun main(arg0, argv) =
let
    fun sort() =
    let
        val gen    = Rand.mkRandom 0w123
        val data   = List.tabulate(100000,
                           (fn _ => gen()))
        val sorted = ListMergeSort.sort (op >) data
    in
        ()
    end
in
    (* SMLofNJ.Internals.GC.messages true; *)
    Compiler.Profile.setTimingMode true;
    sort();
    Compiler.Profile.setTimingMode false;
    Compiler.Profile.report TextIO.stdOut;

    OS.Process.success
end
```

Here are the performance results on my 1GHz Athlon machine.

```
%time cumsec  #call  name
42.85    .18      1  Main.<tempStr>.main.main.sort.sort
26.19    .29      0  Major GC
23.80    .39      0  Minor GC
 9.52    .42 100000  Main.<tempStr>.main.main.sort.sort.data
  .00    .42      1  Main.<tempStr>.main.main
```

This shows the program spent 9% building the list of 100000 random numbers, 42% in the sort, and 50% doing garbage collection of all of that data. Memory usage peaked at 11MB. SML/NJ likes to use lots of heap space to save on garbage collection. I can get some control over the peak heap size by changing the allocation size used by the garbage collector. The default is 256KB. You can change this by adding a command line argument to the SML

runtime in the script that runs the program. For example if I add `@SMLalloc=1024` then the allocation size is 1MB and the peak heap usage goes up to 22MB but the collection time drops to 29%. If I reduce it to 100KB then the peak usage is around 9MB but the collection rises to 64%.

Operating System Information

The `SMLofNJ.SysInfo` structure provides a collection of functions to return the configuration of the compiler and the platform. If you know that it's a Unix system then the Posix API is likely to be available. If you want to know the endian-ness then the target architecture will tell you. In the 110.0.7 version of SML/NJ the `getOSVersion` function does not work. It always returns `"<unknown>"`.

Lazy Suspensions

Lazy suspensions allow you to "memoise" a function. This means that the function is evaluated at most once. On subsequent calls the result from the first call is returned. This could be useful to initialise an imperative data structure only if actually needed at runtime.

In the `getopt` example of the section called *Using a Hash Table* in Chapter 2 the option table was built when the `Global` structure was compiled. It appeared in the heap file. This would be inefficient if the table is very large. Also if the data structure you want to build depends on some parameter supplied at run-time then you need to build the data structure imperatively after the program starts running. You can do this with a reference variable but a suspension is more convenient.

The following example uses the string table structures from the section called *Using a Hash Table* in Chapter 2.

```
structure Table:
    sig
```

```
        val set:    string * string -> unit
        val get:    string -> string option
    end =
struct
    open Common

    type Table = string STRT.hash_table

    val susp = SMLofNJ.Susp.delay(fn () =>
                STRT.mkTable(101, NotFound): Table)

    fun table() = SMLofNJ.Susp.force susp

    fun set (k, v) = STRT.insert (table()) (k, v)
    fun get k      = STRT.find (table()) k
end
```

I've defined a `Table` structure with `get` and `set` functions. I've used an unnamed signature constraint to only export these functions. The value `susp` is built during the compilation of the structure and leaves a suspension in the heap file. This suspension will be forced to a concrete value the first time that either the `get` or `set` functions is called. This will cause the table to be built. The same table will be used by all calls to the `get` and `set` functions which is important since it is updated in place.

The type constraint on the `mkTable` call is needed to fix the type of the table for the suspension. The *value restriction* rule of SML does not allow a value at the level of a structure declaration to have a polymorphic type (i.e. one with an unresolved type variable).

Weak Pointers

Normally the garbage collector deems a heap object to be garbage once all pointers to the object have been deleted. Sometimes it is convenient to retain a pointer to an object while still allowing the object to be collected. For example you may have a cache of objects that you have fetched from some file. If memory becomes tight you may want the objects to be removed from the cache and collected since you can fetch them again if you really need

them. (Unfortunately you can't prioritise the collection. All weakly referenced objects in a generation will be collected).

A weak pointer is a pointer that is ignored by the garbage collector when deciding whether a heap object is garbage. Normal pointers are called strong pointers. Once all of the strong pointers have disappeared the object can be collected. Then all weak pointers to that object are marked invalid to indicate that they now dangle. You can test if the weak pointer is still valid.

Another use for weak pointers is to do some finalisation after the object has been collected. If you can arrange to scan all weak pointers after each collection then you can detect which objects have been collected because their weak pointers will be invalid. You can trigger a scan of the weak pointers with a signal handler for the `sigGC` pseudo-signal. (See the section called *Signals*).

There is a problem with weak pointers and compiler optimisation. Since, with immutable data structures, copy by value and copy by reference are the same, there can be some ambiguity about whether the various pointers are all pointing to the same copy of an object. You should only use weak pointers to reference variables. This ensures that there is no hidden replication of the object pointed to by the reference variable.

Here is a mickey-mouse example that caches the Unix environment variables in a global hash table for faster access. This of course assumes that the environment isn't changed while the program runs (which it probably won't do since there is no `putenv` operation).

```
structure Environ:
    sig
        val get:    string -> string option
    end =
struct
    open Common
    open SMLofNJ.Weak

    type Table = string STRT.hash_table

    val cache: (Table option) ref weak ref = ref (weak (ref NONE))
```

```
    fun table() : Table =
    let
        (*  This takes a NAME=VALUE string *)
        fun fill tbl env =
        let
            val ss = Substring.all env
            val str = Substring.string
            val fields = Substring.fields
                           (fn c => c = #"=") ss
        in
            case fields of
              [n, v] => STRT.insert tbl (str n, str v)
            | [n]    => STRT.insert tbl (str n, "")
            | _      => ()      (* unrecognisable *)
        end

        fun build() =
        let
            val tbl = STRT.mkTable(101, NotFound)
        in
            print "building\n";
            app (fill tbl) (Posix.ProcEnv.environ());
            cache := weak (ref (SOME tbl));
            tbl
        end
    in
        case strong (!cache) of
          NONE => build()       (* has been collected *)

        | SOME rtbl =>
        (
            case !rtbl of
              NONE     => build() (* is not yet built *)
            | SOME tbl => tbl     (* table is available *)
        )
    end

    fun get k = STRT.find (table()) k
end
```

Instead of a suspension as I did in the section called *Lazy Suspensions* I've used a reference variable. With one of those I can have the variable initialised to the `NONE` state so that the table isn't built until called for. The `table` function fetches the table or builds/rebuilds it if it is not available. The `weak` function creates a weak pointer to the reference. The `strong`

function returns the reference if it is still available. Since the type of `strong` is `'a weak -> 'a option` the value in the case expression has the type `(Table option) option` which gives us the three cases. After building the table a weak reference to it is assigned to the cache. Note the extra `ref` between the weak reference in the table. This is just to ensure that we only have weak references to `ref` types.

Here is the main function that I use to test it. I build a big list in between two calls to get an environment variable. This triggers a garbage collection and I can see that the build is done twice. If I comment out the call to `data` then the build only happens once.

```
fun main(arg0, argv) =
let
    fun data() = ignore(List.tabulate(100000, fn n => n))
in
    SMLofNJ.Internals.GC.messages true;
    print(concat["The PATH is ",
                 valOf(Environ.get "PATH"), "\n"]);
    data();
    print(concat["The PATH is ",
                 valOf(Environ.get "PATH"), "\n"]);

    OS.Process.success
end
```

The Exception History List

Getting access to the exception history list is a new feature which has crept in to the compiler in the 110.0 version. It shows the source location where the exception was raised. Here is an example of it in the top-level uncaught exception handler.

```
fun main(arg0, argv) =
let
    fun bad() = raise Fail "bye"
in
    bad();
    OS.Process.success
```

```
end
handle x => (
    toErr(concat["Uncaught exception: ",
                 exnMessage x, " from\n"]);

    app (fn s => (print "\t"; print s; print "\n"))
            (SMLofNJ.exnHistory x);

    OS.Process.failure
    )
```

The Socket API

There is fairly comprehensive API for socket programming in the compiler's Basis library. This is an SML/NJ extension that has gone undocumented until now as far as I know.

You can find the source for the API in the boot/Sockets directory of the compiler source. Start with the `SOCKET` signature in the `socket-sig.sml` file. The actual implementation starts with the shared material in the `PreSock` structure in the `pre-sock.sml` file.

The Generic Socket Types

In the `Socket` structure (with signature `SOCKET`) we have the following generic types.

```
type ('af, 'sock) sock
type 'af sock_addr

(* witness types for the socket parameter *)
type dgram
type 'a stream
type passive    (* for passive streams *)
type active     (* for active (connected) streams *)
```

The clever thing here is the use of type parameters to distinguish between different kinds of sockets. This lets the type checker do some checking of the use of sockets. Internally a socket is represented by the following datatype in the `PreSock` structure.

```
(* the raw representation of a socket
   (a file descriptor for now) *)
type socket = int
datatype ('af, 'sock) sock = SOCK of socket
```

The type just includes the integer file descriptor for the socket. The type variables are not actually used in the definition of a socket. They are only a part of the logical framework of the program that is checked by the type checker at compile time[2].

The first type parameter to `sock` distinguishes the different address families. All of the functions in the Socket structure accept a socket type, such as `('a, 'b) sock`, with any address family, as you would expect. Address families are used at the time sockets are created. See the section called *A Simple TCP Client* for an example. The `Socket.AF` structure defines a type for address families and some functions to obtain values of the type. Normally you would use the specialised types of the section called *The Specific Socket Types*.

The second type parameter `'sock` distinguishes between the different states of a socket. The possible types are:

- `dgram` for a datagram (UDP) socket;
- `passive stream` for a stream (TCP) socket that will be used to accept a connection but has not yet been connected, and
- `active stream` for a connected stream (TCP) socket.

Some functions in `Socket` only operate on passive or active streams. For example

```
val accept: ('a, passive stream) sock ->
            (('a, active stream) sock * 'a sock_addr)
```

```
val listen: (('a, passive stream) sock * int) -> unit

val sendVec: (('a, active stream) sock * Word8Vector.vector buf)
             -> int
```

The type parameters constrain you to ensure that you cannot call `sendVec` on the same socket value that you passed to `accept` or `listen`. You can however call `sendVec` on the value returned from `accept`.

Socket addresses are defined in `Socket` as being generic over address families. But you will use more specific types with their own functions for addresses in a specific family.

The Specific Socket Types

The socket types in the section called *The Generic Socket Types* are generic over the address family. What you will actually use are sets of socket types with the address family fixed. For example the `INetSock` structure defines a type of socket with the address family fixed at `AF_INET` for the internet protocols. The new types and values are:

```
datatype inet = INET

type 'a sock = (inet, 'a) Socket.sock
type 'a stream_sock = 'a Socket.stream sock
type dgram_sock = Socket.dgram sock

type sock_addr = inet Socket.sock_addr
```

Here a distinct type called `inet` has been defined, although it contains no data. Because it is defined with a datatype it is guaranteed to be different from any other type. This allows the type checker to ensure that you don't mix up sockets with different address families. The remaining types are specialisations for the `inet` family. The type variable in the `stream_sock` type will range over the types `passive` and `active` in the `Socket` structure.

The value `INetSock.inetAF` is an address family value, of type `Socket.AF.addr_family`, should you need to specify the family explicitly.

The `UnixSock` structure provides types equivalent to those in `INetSock` but with the address family fixed for Unix domain sockets.

Socket Addresses

The type `sock_addr` represents an address that you can bind a socket to. The generic address, `Socket.sock_addr`, is parameterised by the address family. If you look in the `PreSock` structure you will see that a socket address is represented internally by a byte vector.

```
type addr = Word8Vector.vector
datatype 'af sock_addr = ADDR of addr
```

For each particular address family there is a specialised address type. For example in the `INetSock` structure there is:

```
datatype inet = INET
type sock_addr = inet Socket.sock_addr

val toAddr   : (NetHostDB.in_addr * int) -> sock_addr
val fromAddr : sock_addr -> (NetHostDB.in_addr * int)
val any  : int -> sock_addr
```

The `toAddr` function will coerce an internet address and a port number to a socket address which is specialised for the `inet` address family. The `fromAddr` function will do the reverse. The any function uses the `0.0.0.0` internet address (the traditional `INADDR_ANY`) that you bind a server socket to if you want it to accept connections from any source address. Its argument is the port number.

To lookup an internet address you use the functions in the `NetHostDB` structure. These provide the equivalent of the C library's `gethostbyname` and `gethostbyvalue` functions. The signature for this structure is:

```
signature NET_HOST_DB =
```

```
sig
    eqtype in_addr
    eqtype addr_family
    type entry
    val name     : entry -> string
    val aliases  : entry -> string list
    val addrType : entry -> addr_family
    val addr     : entry -> in_addr
    val addrs    : entry -> in_addr list
    val getByName    : string -> entry option
    val getByAddr    : in_addr -> entry option

    val getHostName : unit -> string

    val scan       : (char, 'a) StringCvt.reader ->
                     (in_addr, 'a) StringCvt.reader
    val fromString : string -> in_addr option
    val toString   : in_addr -> string
end
```

You use the `getByName` or `getByAddr` functions to fetch a database entry, equivalent to C's `struct hostent`. They return NONE if the entry is not found. The functions `name` through to `addrs` fetch the fields of an entry. The `fromString` function will parse an address in the numeric formats *a.b.c.d*, *a.b.c*, *a.b* or *a*. Where there is more than one digit the left digits are 8 bit values and the last digit takes up the rest of the address. Hex numbers are allowed with a `0x` prefix, octal with a `0` prefix.

For the Unix address family you have in the `UnixSock` structure:

```
datatype unix = UNIX
type sock_addr = unix Socket.sock_addr

val toAddr   : string -> sock_addr
val fromAddr : sock_addr -> string
```

The string is the path to the socket in the file system.

A Simple TCP Client

This example program makes a TCP connection to a port and fetches one line of response and prints it. You can test it against a server such as the SMTP mail server on port 25 or the NNTP server on port 119. Here is the central function. It's fairly straightforward.

```
fun connect port =
let
    val localhost =
            valOf(NetHostDB.fromString "127.0.0.1")
    val addr = INetSock.toAddr(localhost, port)
    val sock = INetSock.TCP.socket()

    fun call sock =
    let
        val _    = Socket.connect(sock, addr)
        val msg  = Socket.recvVec(sock, 1000)
        val text = Byte.bytesToString msg
    in
        print text;
        Socket.close sock
    end
    handle x => (Socket.close sock; raise x)
in
    call sock
end
handle OS.SysErr (msg, _) => raise Fail (msg ^ "\n")
```

The `recvVec` function performs the C library `recv()` on the socket into a buffer of 1000 bytes. Since we are expecting a text response the `bytesToString` coerces the byte vector into a text string. I've wrapped the connection phase into a function to make it easier to wrap an exception handler around it. The handler closes the socket and reraises the exception. This is overkill for such a simple program but it shows you what you would need to do in a larger program. All errors from the socket functions raise `OS.SysErr` exceptions. The exception handler for these translates them into a simpler error message.

Here is the main program to call the `connect` function.

```
fun toErr msg = TextIO.output(TextIO.stdErr, msg)
```

```
fun main(arg0, argv) =
let
in
    case argv of
      [port] =>
        (case Int.fromString port of
          NONE => raise Fail "Invalid port number\n"

        | SOME p => connect p)

    | _ => raise Fail "Usage: simpletcp port\n";

    OS.Process.success
end
handle
  Fail msg => (toErr msg; OS.Process.failure)

| x =>
(
    toErr(concat["Uncaught exception: ",
                 exnMessage x, " from\n"]);
    app (fn s => (print "\t"; print s; print "\n"))
        (SMLofNJ.exnHistory x);
    OS.Process.failure
)
```

A Simple TCP Server

This example program complements the simple client of the previous section. It listens on a TCP socket and sends a simple text response to each client that connects. It is a single threaded server. Here is the `serve` function that runs the server.

```
fun serve port =
let
    fun run listener =
    let
        fun accept() =
        let
            val (conn, conn_addr) = Socket.accept listener
        in
```

```
            respond conn;
            accept()
        end

        and respond conn =
        let
            val msg = "hello world from tcpserver\n"
            val buf = {buf = Byte.stringToBytes msg,
                       i = 0, sz = NONE}
        in
            ignore(Socket.sendVec(conn, buf));
            Socket.close conn
        end
        handle x => (Socket.close conn; raise x)

    in
        Socket.Ctl.setREUSEADDR(listener, true);
        Socket.bind(listener, INetSock.any port);
        Socket.listen(listener, 9);
        accept()
    end
    handle x => (Socket.close listener; raise x)
in
    run (INetSock.TCP.socket())
end
handle OS.SysErr (msg, _) => raise Fail (msg ^ "\n")
```

Again I have used functions to isolate the scope of exception handlers as well as to implement the server loop. The `run` function sets up the socket to listen for connections and runs a loop to accept each one. The socket is bound to a given port but its address is set to `0.0.0.0` (`INADDR_ANY`) to accept from any host. The `listen` function takes an integer backlog parameter, the same as the C library `listen()` function.

Each accepted connection returns a new socket, called `conn`, and the address of the connecting peer which I ignore. The `respond` function builds a buffer to send to the client. The `sendVec` function performs the C library `send()` function and returns its result which will be the number of bytes successfully sent. In this simple server I ignore this. If there is actually an error then the `OS.SysErr` exception will be raised. The buffer argument to `sendVec` must be a record with this type:

```
type 'a buf = {buf : 'a, i : int, sz : int option}

val sendVec: (('a, active stream) sock * Word8Vector.vector buf)
             -> int
```

The type variable `'a` will be either a vector or an array of bytes depending on the function you use. The signature of the `sendVec` function is shown. The `i` field is the offset into the buffer where the send is to start. The `sz` field is the optional length of the data to send. If it is `NONE` then the data extends to the end of the buffer. The standard `Subscript` exception is raised if the offset and length don't fit into the buffer.

The main function for this program is almost identical to the client. It just gets a port number from the command line.

Servers with Multiple Connections

If you want to write a server to handle multiple connections then you can either write it in a single-threaded manner using the poll functions in the `OS.IO` structure or you can use the Concurrent ML library for a more multi-threaded style.

To use polling you will need the `Socket.pollDesc` function:

```
val pollDesc : ('a, 'b) sock -> OS.IO.poll_desc
```

This will obtain a descriptor from the socket suitable for use with `OS.IO`. Here is some example code for polling a set of sockets for reading or writing.

```
type ServerSock = Socket.active INetSock.stream_sock

datatype Handler = Handler of {
        socket: ServerSock,
        reader: ServerSock -> unit,
        writer: ServerSock -> unit
        }

fun poll (handlers: Handler list) =
let
```

```
    (*  Convert to a list annotated with iodesc. *)
    fun to_iodesc (Handler {socket, reader, writer}) =
        (OS.IO.pollToIODesc(Socket.pollDesc socket),
            socket, reader, writer)

    val with_iodesc = map to_iodesc handlers

    (*  Generate a list of poll descriptors for reading
        and writing.
    *)
    fun to_poll (Handler {socket, ...}) =
            (OS.IO.pollIn o OS.IO.pollOut o Socket.pollDesc)
            socket

    (*  Search for the matching handlers. *)
    fun check_info poll_info =
    let
        val info_iodesc = OS.IO.pollToIODesc(
                                OS.IO.infoToPollDesc poll_info)
        val handler = List.find
                      (fn arg => (#1 arg) = info_iodesc)
                      with_iodesc
    in
        case handler of
          NONE => raise Fail "polled a non-existent socket!"

        | SOME (iodesc, socket, reader, writer) =>
        (
            if OS.IO.isIn  poll_info then reader socket else ();
            if OS.IO.isOut poll_info then writer socket else ()
        )
    end

    val info_list = OS.IO.poll(map to_poll handlers, NONE)
in
    app check_info info_list
end
```

I've defined a record type for a handler that maps a socket to reader and writer functions. These functions will be called when the socket is ready for reading or writing respectively. My `poll` function takes a list of handlers and calls the readers and writers for each socket that is ready for I/O. The first step is to extend the handler data with an `OS.IO.iodesc` value. This is the only type of value used in `OS.IO` that supports the equality operator so

that I can use it for looking up the handler. The `Socket` structure only provides for producing an `OS.IO.poll_desc` which I have to back-convert to an `iodesc`.

The `to_poll` function separately converts each socket to a `OS.IO.poll_desc` type. The `pollIn` and `pollOut` mark the descriptor for polling for input and output respectively. I then pass the descriptors to the `OS.IO.poll` function to get the list of resulting info records in `info_list`. I'm not using a timeout here.

The `check_info` function examines each info record. First I extract the `iodesc` from the info record. Then I search the `with_iodesc` list for a record with the same `iodesc`. The argument to the predicate is an annotated tuple. I use the `#1` notation to get the first member of the tuple which is the `iodesc`. The `isIn` function tests if the info record indicates a socket ready for reading. If so then I call the reader. Similarly for the writer.

Here is part of the modified `serve` function from the server in the section called *A Simple TCP Server*. It's just a trivial example of calling the poll function.

```
fun serve port =
let
    fun run listener =
    let
        fun accept() =
        let
            val (conn, conn_addr) = Socket.accept listener
        in
            poll [Handler {
                    socket = conn,
                    reader = reader,
                    writer = writer
                    }];
            accept()
        end

        and writer conn =
        let
            val msg = "hello world from tcpserver\n"
            val buf = {
                    buf = Byte.stringToBytes msg,
```

```
            i = 0,
            sz = NONE
            }
in
    print "responding to a client\n";
    ignore(Socket.sendVec(conn, buf));
    Socket.close conn
end
handle x => (Socket.close conn; raise x)

and reader conn = ()
```

A serious server would need to maintain a data structure of current connections. This might be a list of records similar to `Handler`. However you will get a nicer result if you use Concurrent ML to write the server in a multi-threaded style. This will have a reader and a writer thread for each connection. See Chapter 6.

Notes

1. There should be a `Unsafe.Real64Vector` but it isn't implemented yet.
2. I've corrected the order of the type variables which is a typo in `PreSock.sock` that has no effect.

Chapter 5. The Utility Libraries

SML/NJ comes with a collection of utility libraries that are separate from the compiler. These provide modules for the following main areas.

- Data structures such as hash tables, dynamically resizing arrays, bit vectors, queues, maps and trees and property lists.
- Algorithms such as list sorting, searching, numeric formatting, random numbers.
- Utility modules for simple parsing, and miscellaneous I/O operations.
- Regular expressions.
- Parsing HTML 3.2 markup.
- Some Unix and socket utilities.

You can find these in the `smlnj-lib` source directory. At the moment these modules are underdocumented. There is some documentation in the `Doc` directory but it is mainly reference material which describes the API. For full details you will need to look through the source files.

In the following sections I will describe the modules briefly and give examples for those that I think will be the most useful.

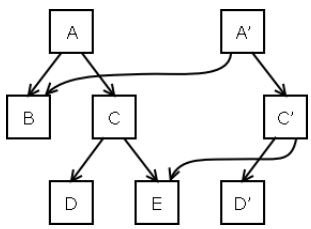
Data Structures

Trees, Maps and Sets

The major data structure (outside of lists) is the tree. The utility library provides an implementation of red-black trees with functional behaviour. This means that an update of a tree produces a new tree leaving the original unchanged.

You might think that producing a new updated tree would be expensive as it seems to require copying the whole tree. But that copying is mostly by reference. Figure 5-1 shows what you get after updating a functional binary tree. In this diagram the node D has been replaced with a new node D'. This requires that the parent node C be updated with the new reference to D'. The updates propagate up the tree to the root at A' each producing a copy of the updated node. But no more nodes will be copied than the height of the tree. All other nodes will be shared. If the application drops the reference to node A then the garbage collector will reclaim the old nodes A, C and D. In the mean-time the application has access to both the old and new versions of the tree which can be useful (e.g. for undoing the update).

Figure 5-1. Updating a Tree.



As long as the tree stays reasonably balanced then the height of the tree stays minimal and lookups and updates are cheapest. The red-black algorithm adjusts the tree after each update to keep it close to balanced. The algorithm is fairly complex. A generic implementation is provided in the `RedBlackMapFn` functor in the `redblack-map-fn.sml` source file.

```
functor RedBlackMapFn (K : ORD_KEY) :> ORD_MAP where Key = K =
...

signature ORD_KEY =
sig
    type ord_key

    val compare : ord_key * ord_key -> order
end
```

```
...
datatype order = LESS | EQUAL | GREATER
```

This functor can be specialised for a key of any ordered type. The `ORD_KEY` signature describes the key type. The `order` type is predefined in the SML Basis. The resulting structure satisfies the `ORD_MAP` signature defined in the `ord-map-sig.sml` file. This signature describes the tree operations such as `insert`, `remove` etc.

The library then goes on to use red-black trees to implement sets. A set is just a map from a domain to a boolean value (which is always true for members of the set). But for efficiency a separate implementation of red-black sets is provided in the `RedBlackSetFn` functor.

Next the library provides some specialisations of these red-black sets and maps for various keys. For keys of `int` and `word` the code is re-implemented, presumably for efficiency. For other key types such as atoms the functor is specialised.

```
structure AtomRedBlackMap =
  RedBlackMapFn (
    struct
      type ord_key = Atom.atom
      val compare = Atom.compare
    end)
```

An atom is a string paired with a hash value so that equality can be tested very quickly by first comparing the hash values. The library's implementation of atoms also ensures that all equal strings share the same atom by reference. They are useful in the symbol tables of compilers and any similar application.

Other kinds of maps are also implemented. The `BinaryMapFn` functor in the `binary-map-fn.sml` source file also keeps the tree reasonably balanced. The implementation says "The main advantage of these trees is that they keep the size of the tree in the node, giving a constant time size operation." Matching implementations for sets and integer and atom keys are provided. You can also try out the `SplayMapFn` functor which does it with splay trees

and the `ListMapFn` functor which does it with sorted lists. These all conform to the `ORD_MAP` and `ORD_SET` signatures so they are interchangable.

To demonstrate functional maps here is a test program that reads pairs of numbers from a file and makes a map of them. The map is then printed out.

```
structure Main=
struct
    structure Map = IntRedBlackMap

    fun toErr s = TextIO.output(TextIO.stdErr, s)


    fun read_file file : int Map.map =
    let
        val istrm = TextIO.openIn file

        (*  Read a pair of ints on a line and loop.
            Empty lines are ignored. Other junk is fatal.
        *)
        fun get_pairs map_in lnum =
        (
            case TextIO.inputLine istrm of
              "" => (TextIO.closeIn istrm; map_in)   (* eof *)

            | line =>
            let
                val tokens = String.tokens Char.isSpace line
            in
                case map Int.fromString tokens of
                  [] => get_pairs map_in (lnum+1)

                | [SOME a, SOME b] =>
                      get_pairs (Map.insert(map_in, a, b)) (lnum+1)

                | _ => raise Fail (concat["Invalid data on line ",
                                          Int.toString lnum])
            end
        )
        handle x => (TextIO.closeIn istrm; raise x)

    in
        get_pairs Map.empty 1
    end
```

```
    and show_pairs pairs =
    let
        fun show (a, b) = print(concat[
            Int.toString a, " => ", Int.toString b, "\n"])
    in
        Map.appi show pairs
    end


    fun main(arg0, argv) =
    (
        case argv of
          [file] =>
        (
            show_pairs(read_file file);
            OS.Process.success
        )

        | _ =>
        (
            print "Usage: intmap file\n";
            OS.Process.failure
        )
    )
    handle x =>
    (
        toErr(exnMessage x); toErr("\n");
        OS.Process.failure
    )

    val _ = SMLofNJ.exportFn("intmap", main)
end
```

Hash Tables

A very different kind of map is the hash table. The implementation is imperative meaning that the table is updated in place. See the section called *Using a Hash Table* in Chapter 2 for an example of using a hash table to map from strings to strings.

The hash function that is used is defined in the `HashString` structure in the `hash-string.sml` file. It implements the recursion `(h = 33 * h + 720`

`+ c)` over the characters of the string. The same hash function is used for atoms.

Vectors and Arrays

The utility library provides three useful kinds of arrays in addition to the array types defined in the Basis library (see the section called *Arrays and Vectors* in Chapter 3).

- There is a monomorphic dynamic array functor which automatically grows in size to accomodate the data.
- There is a `BitArray` structure that stores bits compactly in bytes. It provides all of the standard array operations. In addition there are bit-operations like `and`, `or` and `shift` over arrays. See the `bit-array.sml` source file for more details. There is a matching `BitVector` structure for immutable bit vectors.
- There is an `Array2` structure for polymorphic two-dimensional arrays. See the `array2.sml` source file for more details.

The dynamic array grows by at least doubling in size as needed to accomodate all of the elements that have been referenced. This doubling requires copying the original array. The elements are stored internally in a reference to an array so the growth is imperative. Newly created elements are initialised with a default value. The module is a functor which constructs the dynamic array from a monomorphic array.

```
functor DynamicArrayFn (A : MONO_ARRAY) : MONO_DYNAMIC_ARRAY

signature MONO_DYNAMIC_ARRAY =
sig
    type elem
    type array

    val array:      (int * elem) -> array
    val subArray:   array * int * int -> array
    val fromList:   elem list * elem -> array
```

```
    val tabulate:   int * (int -> elem) * elem -> array
    val default:    array -> elem
    val sub:        array * int -> elem
    val update:     array * int * elem -> unit
    val bound:      array -> int
    val truncate:   array * int -> unit
end
```

The signature `MONO_ARRAY` is predefined in the Basis library. It characterises any type that can be made to behave like an imperative array (see the section called *Arrays and Vectors* in Chapter 3). The `MONO_DYNAMIC_ARRAY` signature provides a restricted set of operations on dynamic arrays which currently omits the iterator operations. See the source file `mono-dynamic-array-sig.sml` for more details on the operations.

Here is an example of creating a dynamic array of 1000 real numbers initialised to zero (the default value).

```
structure DynRealArray = DynamicArrayFn(Real64Array)

val reals = DynRealArray.array(1000, 0.0)
```

There is a `MonoArrayFn` functor which is a little utility for creating array structures for `DynamicArrayFn`. For example, since there is no predefined `IntArray` structure you could write

```
structure DynIntArray = DynamicArrayFn(MonoArrayFn(type elem = int))
```

Queues and Fifos

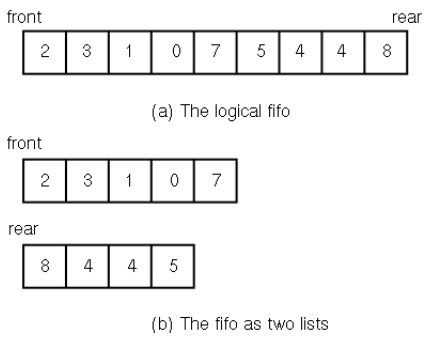
The utility library implements first-in-first-out queues in both the functional and imperative styles. The functional ones are called fifos and the imperative ones are called queues (for no special reason).

The fifo implementation is in the `Fifo` structure in the `fifo.sml` source file. The queue implementation is in the `Queue` structure in the `queue.sml` source file. It's actually just a wrapper for a fifo stored in a reference.

The implementation of fifos is a little tricky for a typical functional language like SML. It requires access to both ends of a list. But SML only provides cheap access to the front of a list. If you naively enqueued a value by appending it to the end of a list that would require copying the entire list which would be ridiculously expensive.

The solution is to split the list into front and rear halves with the rear half in reverse. This moves the end of the fifo to the front of a list, as shown in Figure 5-2.

Figure 5-2. A Fifo as a Pair of Lists.



Elements can be dequeued from the front of the fifo by just removing them from the front of the 'front' list. Elements can be enqueued to the rear of the fifo by adding them to the front of the 'rear' list. When the front list becomes empty then the elements in the rear list are transferred to it. This still requires copying a list but much less often than the naive implementation above. A detailed analysis of the performance of this solution can be found in [*Okasaki*]. It turns out that the average time to enqueue or dequeue an element is O(1), that is, effectively constant.

Property Lists

A property list (*plist*) is a list of key-value pairs that can be attached to or associated with some value. The Lisp ([*Steele*]) language has had plists since its beginning in 1959. In Common Lisp they are only used for annotating symbols. The SML design allows you to annotate any base value such as a node in a tree. You can add or remove properties from the list so they are naturally imperative.

The utility library has an implementation of property lists in the `PropList` structure of the `plist.sml` source file. A key can appear only once in a property list but it can appear in any number of property lists. In Lisp the keys would typically be Lisp *symbols* but in SML any type that supports equality will do. The design provides a more abstract interface to a property. Each property is represented by a group of functions that access the property. The actual implementation of the key is an internal detail. Here is the signature of the `PropList` structure.

```
signature PROP_LIST =
sig
    type holder

    val newHolder : unit -> holder

    val clearHolder : holder -> unit

    val sameHolder : (holder * holder) -> bool
        (* returns true, if two holders are the same *)

    (* newProp (selHolder, init) *)
    val newProp : (('a -> holder) * ('a -> 'b)) -> {
            peekFn : 'a -> 'b option,
            getFn  : 'a -> 'b,
            setFn  : ('a * 'b) -> unit,
            clrFn  : 'a -> unit
          }

    val newFlag : ('a -> holder) -> {
            getFn : 'a -> bool,
            setFn : ('a * bool) -> unit
          }
end
```

A `holder`, unsurprisingly, holds one property list. To associate a property list with a value you must be able to write a function that maps from the value to a holder. This function is called `selHolder` in the signature. How you write this is up to you. For example if you were attaching property lists to nodes in a tree you would simply include a holder as one of the fields of a node. The `selHolder` function would then just select the field from a node. The `'a` type variable represents the type of the annotated value and `'b` is the type of the property's value.

The `newHolder` function creates a new empty holder. The `clearHolder` function deletes all of the properties in a holder.

The `newProp` function defines a new property. The property is represented by the four access functions: peek, get, set and clear. These operate in terms of the annotated value so you have to supply the `selHolder` function to translate the annotated value to the holder. The property is restricted to appear only in plists that can be selected by the `selHolder` function. This usually means it is restricted to the plists of one annotated type.

The `init` function is used to create an initial value for the property should you try to get the property value before it is set. This initial value is allowed to depend on the annotated value for more flexibility.

The `newFlag` function makes a specialised kind of property that has no value. You only test if it is present or absent in the plist. The get function returns true if it is present.

Here is a simple demonstration of some properties. Where this demonstration differs from other kinds of data structures is that the set of properties is completely dynamic. You can invent new properties on the fly rather than just having the fixed number of fields in a record. Accessing these properties will be a bit slow though. First I define a set of people and some properties for them.

```
structure PL = PropList

(*  Associate a plist holder with each person. *)
val people: PL.holder STRT.hash_table = STRT.mkTable(101, NotFound)
```

```
(*  Add someone to the table. *)
fun define name = STRT.insert people (name, PL.newHolder())

(*  Define some properties.
    Weight is a real measure in kilograms. Father is a string.
*)
val weight_prop = PL.newProp (STRT.lookup people, fn _ => 0.0)
val father_prop = PL.newProp (STRT.lookup people, fn _ => "unknown")

(*  Functions to set and get the properties. *)
fun set prop (name, value) =
let
    val {peekFn, getFn, setFn, clrFn} = prop
in
    setFn(name, value)
end


fun get prop name =
let
    val {peekFn, getFn, setFn, clrFn} = prop
in
    getFn name
end
```

The people are represented by a hash table that maps a name to a plist holder. (See the section called *Using a Hash Table* in Chapter 2 for details of my `STRT` structure). The `set` and `get` functions are polymorphic enough to work for all properties. Unfortunately the type of a property is not named in the `PROP_LIST` signature so I end up having to write out all of the fields in the property record each time.

Here is a demonstration of the use of these functions. I define some people, set some properties and then dump them all.

```
val names = ["fred", "wilma", "barney", "betty", "wilma",
             "pebbles", "bambam"]


fun show_father name = print(concat[
                        name, "\thas father ",
                        get father_prop name,
                        "\n"])
```

```
fun show_weight name = print(concat[
                        name, "\thas weight ",
                        Real.toString(get weight_prop name),
                        "kg\n"])

fun run() =
let
in
    app define names;

    app (set father_prop) [("pebbles", "fred"),
                           ("bambam", "barney")
                           ];

    app (set weight_prop) [("fred", 111.0),
                           ("wilma", 73.0),
                           ("barney", 82.5),
                           ("betty", 68.5),
                           ("pebbles", 15.1),
                           ("bambam", 18.7)
                           ];
    app show_father names;
    app show_weight names;
    ()
end
```

What is especially interesting about SML plists is how they are implemented. A list in SML must always have elements of the same type. But property lists manage to have elements of different types and new types of elements can be added at any time. How do they do it?

If you were to implement something like this in an object-oriented language such as C++ or Java you would define a base class for a plist element. Then you would define a subclass for each type of element you want in the list. You can use run-time type identification ("downcasting") to extract a property's value. A subclass defines a subtype of the base class and you can use any subtype anywhere that its base type is used. This is how you do polymorphism in object-oriented languages.

But SML does not provide subtyping. Normally you have to define a datatype that represents the union of all of the subtypes you are going to use. This doesn't have the flexibility or extensibility of the object-oriented paradigm.

But there is a way around this. There is a dirty little trick you can do with exceptions in SML that provides a great big loophole in the type checking. An exception declaration declares a constructor that coerces some type to the special type `exn`. For example this declaration

```
exception Error of string
```

declares the constructor function

```
val Error: string -> exn
```

You can think of the `exn` type as being like a datatype where each exception declaration adds a new branch, no matter where the declaration appears in your program. The `exn` type is an extensible type. From another point of view, the `Error` declaration above lets you use a `string` type anywhere an exception type is allowed so it effectively declares the `string` type as a subtype of `exn`.

Exceptions have one more quirk up their sleeve. An exception declaration is, to use the jargon, *generative*. This means that each declaration creates a new exception each time that it is executed. For example if you have an exception declaration inside a function then each time that function is called a new distinct exception is created, even though the constructor name is the same. This is what lets you define new subtypes dynamically.

Here is an example where the properties are defined statically using exceptions.

```
type PList = exn list

exception PFruit of string
exception PNum   of int

val fruit = [PFruit "apple", PNum 5]

fun get_fruit []            = NONE
|   get_fruit (PFruit f::_) = SOME f
|   get_fruit (_::rest)     = get_fruit rest
```

The list `fruit` contains some properties. The `get_fruit` will find and return the value of the `PFruit` property. This all works the same way as if I had written

```
datatype PList =
        PFruit of string
    |   PNum   of int
```

This next example creates properties dynamically.

```
fun new_prop dflt =
let
    exception E of 'a

    fun get []        = NONE
    |   get (E v::_)  = SOME v
    |   get (_::rest) = get rest

    fun set props v = (E v)::props

    fun dummy() = E dflt
in
    (get, set)
end

val (get_colour, set_colour) = new_prop "colour"
val props2       = set_colour fruit "red"

val (get_weight, set_weight) = new_prop 0.0
val props3       = set_weight props2 0.75

fun report() =
(
    print(concat[
        "The colour is ", valOf(get_colour props3),
        " and the weight is ",
        Real.toString(valOf(get_weight props3)),
        "\n"])
)
```

Every time the `new_prop` function is called the exception declaration will be executed. This will define a new exception which is known locally by the name `E`. This new exception is captured by the `get` and `set` functions which provide the only access to it. I've had to include an extra dummy argument

to let me constrain the type of the value for each property. Without it, the type `'a` is completely unconstrained and the type checking fails with the notorious value restriction message:

```
dyn.sml:36.5-36.34 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
dyn.sml:37.5-37.34 Error: operator and operand don't agree [tycon mismatch]
  operator domain: ?.X1
  operand:         string
  in expression:
    (set2 fruit) "red"
```

By including the `dflt` argument and the `dummy` function I constrain the type of the property value to be the same as that of the `dflt` argument. So when I write `(new_prop "colour")` the `get_colour` and `set_colour` functions are known to work on strings. In a more complete example the `dflt` argument would be useful as a default or initial value for the property.

With this exception trick and property lists you can build up a nice little prototype-based object system along the lines of the Common Lisp Object System (CLOS)[*Steele*].

Algorithms

This section describes the algorithms in the Util directory of the SML/NJ library.

Sorting and Searching

The utility library provides a functional polymorphic merge-sort for lists. It is in the `ListMergeSort` structure of the `list-mergesort.sml` source file. Here is the signature.

```
signature LIST_SORT =
sig
```

```
     val sort : ('a * 'a -> bool) -> 'a list -> 'a list
        (* (sort gt l) sorts the list l in ascending order using the
         * "greater-than" relationship defined by gt.
         *)

     val uniqueSort : ('a * 'a -> order) -> 'a list -> 'a list
       (* uniquesort produces an increasing list, removing equal
        * elements
        *)

     val sorted : ('a * 'a -> bool) -> 'a list -> bool
        (* (sorted gt l) returns true if the list is sorted in ascending
         * order under the "greater-than" predicate gt.
         *)
end
```

Here is an example of their use.

```
val strings = [ "fred",
                "wilma",
                "barney",
                "betty",
                "wilma",
                "pebbles",
                "bambam"
            ]


fun sort_strings (data: string list) =
(
    ListMergeSort.sort (op >) data
)


fun unique_strings (data: string list) =
(
    ListMergeSort.uniqueSort String.compare data
)
```

The expression `(op >)` has an ambiguous overloaded type. The compiler has to be able to decide the type at compile time so it can't be polymorphic over all types with a greater-than operator. In `sort_strings` the type is fixed at string comparison by the context. I could have been more explicit and written `String.>` for the operator.

The `unique_strings` function removes duplicate strings from the list. The family of compare functions for the basic types return the three-way comparison: `LESS`, `EQUAL` or `GREATER`.

If you are working with arrays there is a quick-sort implementation in `ArrayQSort` with the following signature. It sorts an array in-place so it's imperative. It uses the compare functions such as the `String.compare` I mentioned above.

```
signature ARRAY_SORT =
sig
    type 'a array

    val sort   : ('a * 'a -> order) -> 'a array -> unit
    val sorted : ('a * 'a -> order) -> 'a array -> bool
end
```

A more abstract version of this for any array-like types is provided by this functor.

```
functor ArrayQSortFn (A : MONO_ARRAY) : MONO_ARRAY_SORT

signature MONO_ARRAY_SORT =
sig
    structure A : MONO_ARRAY

    val sort:    (A.elem * A.elem -> order) -> A.array -> unit
    val sorted:  (A.elem * A.elem -> order) -> A.array -> bool
end
```

Once you've sorted your array you can do a binary search on it.

```
functor BSearchFn (A : MONO_ARRAY) : sig

    structure A : MONO_ARRAY

    (*  binary search on ordered monomorphic arrays. The comparison
        function cmp embeds a projection function from the element type
        to the key type.
    *)
    val bsearch : (('a * A.elem) -> order)
          -> ('a * A.array) -> (int * A.elem) option
    end
```

The first argument to `bsearch` is the comparison function. It compares the key you are searching for with an element in the array. For example the array element might be a pair and you want to compare the key against the first element of the pair. The second is a pair of the key to search for and the array. If the array element is found then its index and value is returned. Here is an example of table lookup using binary search. The table is "static" i.e. built and sorted before being saved to the heap.

```
local
    datatype Gender = Male | Female

    type Pair = string * Gender

    (*  Compare two pairs. *)
    fun compare ((n1, _), (n2, _)) = String.compare(n1, n2)

    structure PairArray = MonoArrayFn(type elem = Pair)
    structure Searcher  = BSearchFn(PairArray)
    structure Sorter    = ArrayQSortFn(PairArray)

    val gender = [  ("fred",        Male),
                    ("wilma",       Female),
                    ("barney",      Male),
                    ("betty",       Female),
                    ("wilma",       Female),
                    ("pebbles",     Female),
                    ("bambam",      Male)
                ]

    val sorted_gender = PairArray.fromList gender
    val _ = Sorter.sort compare sorted_gender
in
    fun find_gender name =
    let
        (* Compare a key with a pair *)
        fun cmp (key, (n, _)) = String.compare(key, n)
    in
        case Searcher.bsearch cmp (name, sorted_gender) of
          NONE             => NONE
        | SOME (_, (_, g)) => SOME g
    end


    fun show_gender Male   = "male"
```

```
    |   show_gender Female = "female"
end
```

Since `BSearchFn` needs a `MONO_ARRAY` I have to be consistent and use `ArrayQSortFn` too. The following code, for example, doesn't work.

```
val sorted_gender = Array.fromList gender
val _ = ArrayQSort.sort compare sorted_gender

Error: case object and rules don't agree [tycon mismatch]
  operator domain: string * PairArray.array
  operand:         string * (string * Gender) array
  in expression:
    (Searcher.bsearch cmp) (name,sorted_gender)
```

The issue is that `PairArray.array` is an opaque type of unknown implementation whereas `ArrayQSort.sort` works on the specialised type `Pair array`. The two are different types as far as the compiler can tell.

Formatted Strings

The utility library provides a `format` function which emulates the C `sprintf` function. It appears in the `Format` structure of the `format.sml` source file. Here is the signature.

```
signature FORMAT =
sig
    datatype fmt_item
      = ATOM of Atom.atom
      | LINT of LargeInt.int
      | INT of Int.int
      | LWORD of LargeWord.word
      | WORD of Word.word
      | WORD8 of Word8.word
      | BOOL of bool
      | CHR of char
      | STR of string
      | REAL of Real.real
      | LREAL of LargeReal.real
      | LEFT of (int * fmt_item)  (* left justify in field *)
      | RIGHT of (int * fmt_item) (* right justify in field *)
```

```
    exception BadFormat         (* bad format string *)
    exception BadFmtList        (* raised on type mismatch *)

    val format  : string -> fmt_item list -> string
    val formatf : string -> (string -> unit) -> fmt_item list -> unit
end
```

The first argument to the `format` function is a printf-style format string. In place of the C varargs mechanism your values to be printed must be wrapped in the `fmt_item` datatype. The `formatf` function can be used to print the string as it is being formed by making its second argument the `TextIO.print` function.

The formats that are recognised have the format

```
"% <flags> <width> <prec> <type>"
```

(without the white space) or "%%" for a literal percent character. The flags are listed in Table 5-1. You can have more than one flag. The width is a decimal integer. The precision value is only allowed for real number formats.

Table 5-1. Format flags.

" "	A blank character means put a blank in the sign field for positive numbers. Negative signs will appear as usual.
+	Put a plus sign for positive numbers.
-	Put a minus sign for negative numbers.
~	Put a tilde for negative numbers. This includes any exponent.
#	Include a base indicator. This means "0" for octal numbers and "0x" for hexadecimal numbers.
0	Pad the number with zeros on the left.

The type characters are listed in Table 5-2.

Table 5-2. Format types.

d	Signed decimal.
X	Uppercase hexadecimal.
x	Lowercase hexadecimal.
o	Octal.
c	Char.
s	String.
b	Bool.
E	Scientific notation with an uppercase exponent.
e	Scientific notation with a lowercase exponent.
f	Floating point.
G	Automatic choice of E or f.
g	Automatic choice of e or f.

Here is a simple example.

```
structure F = Format

fun test_format() =
(
    F.formatf
        "A decimal %d, some hex %#08x and some real %.4f\n"
        print
        [F.INT ~23, F.WORD 0wxbeef, F.REAL 3.14159265]
)
```

It produces this output.

```
A decimal -23, some hex 0x00beef and some real 3.1416
```

Note that the `0x` is counted in the width of the hexadecimal field but that's they way it happens in the C `printf` too.

To go with the format function there is a scan function, in the Scan structure. Here is the signature.

```
signature SCAN =
sig
    datatype fmt_item
      = ATOM of Atom.atom
      | LINT of LargeInt.int
      | INT of Int.int
      | LWORD of LargeWord.word
      | WORD of Word.word
      | WORD8 of Word8.word
      | BOOL of bool
      | CHR of char
      | STR of string
      | REAL of Real.real
      | LREAL of LargeReal.real
      | LEFT of (int * fmt_item)        (* left justify in field *)
      | RIGHT of (int * fmt_item)       (* right justify in field *)

    exception BadFormat                 (* bad format string *)

    val sscanf : string -> string -> fmt_item list option
    val scanf  : string -> (char, 'a) StringCvt.reader
          -> (fmt_item list, 'a) StringCvt.reader

end
```

Although it is not obvious, the `fmt_item` type in the `Scan` structure is the same one as in the `Format` structure, not just different types with the same name. So you can use them interchangably. In the current implementation flags and field widths in the format string are ignored.

The `scanf` function is designed to work with the StringCvt scanning infrastructure (see the section called *Text Scanning* in Chapter 3). The `sscanf` function is just defined as scanf applied to strings using `StringCvt.scanString`. If the return value is NONE then the scan failed. Here is a simple function to test `sscanf`.

```
structure S = Scan

fun test_scan() : unit =
let
    val items = valOf(S.sscanf "%d %s %f" "123 abc 3.45")

    val display = ListFormat.fmt {
                    init  = "[",
```

```
                    sep   = " ",
                    final = "]",
                    fmt   = show_item
                    } items
in
    print display; print "\n"
end


and show_item (S.INT n)  = Int.toString n
|   show_item (S.STR s)  = s
|   show_item (S.REAL r) = Real.toString r
|   show_item _          = "unknown"
```

This example also demonstrates a use of the utilities in the `ListFormat` structure. See the `list-format-sig.sml` source file for more details.

Here is a demonstration of `scanf`. It will continue to read until it finds three integers separated by white space, even over several lines. Any other input will result in failure.

```
fun test_scan_io() =
let
    val _ = print "Enter 3 integers\n"
in
    case TextIO.scanStream (S.scanf "%d %d %d") TextIO.stdIn of
      SOME items => (
        print "got ";
        print (ListFormat.listToString show_item items);
        print "\n"
        )

    | NONE => print "The reading failed\n"
end
```

Miscellaneous Utilities

The recommended random number generator is `Random` in the `random.sml` source file. According to the blurb in the source file it uses *a subtract-with-borrow (SWB) generator as described in Marsaglia and*

Zaman, "A New Class of Random Number Generators," Ann. Applied Prob. 1(3), 1991, pp. 462-480. Here is an extract from the signature.

```
signature RANDOM =
sig
    type rand
        (* the internal state of a random number generator *)

    val rand:       (int * int) -> rand
        (* create rand from initial seed *)

    val toString:   rand -> string
    val fromString: string -> rand

    val randInt:    rand -> int
        (* generate ints uniformly in [minInt,maxInt] *)
```

A generator has the type `rand`. You can create as many generators as you like. A generator is updated imperatively by functions like `randInt`. The `toString` and `fromString` functions would be useful to save the state of a generator in a file.

The `IOUtil` structure in `io-util.sml` contains some functions which perform a work function with the standard input or output redirected to a file. They match some utility functions available in the Scheme language. The functions in the `PathUtil` structure in `path-util.sml` search for files in lists of directories in the Unix PATH format.

The `Iterate` structure in `iterate.sml` provides some simple functions for looping by performing a function multiple times. It includes a generic "for" loop, in case you're hankering for one.

The `TimeLimit` structure in `time-limit.sml` provides a function to perform a work function and interrupt it if it runs for too long. It uses the SML/NJ interval timer facility (see the section called *The Interval Timer* in Chapter 4) which uses the `SIGALRM` signal.

Don't use `TimeLimit` with CML

Because it uses SIGALRM, which is the same signal that the CML library uses, it will ruin the pre-emption of threads.

The remaining few structures in the utility library are oriented towards compiler writers. The `GraphSCCFn` functor in `graph-scc.sml` is a strongly-connected components algorithm for finding cycles in directed graphs. The "uref" source files provide special-purpose reference types that look like they would be useful for type-checking algorithms in compilers. The `ParserComb` structure in `parser-comb.sml` provides some utility functions for hand-written recursive-descent parsers but ML-Lex and ML-Yacc would probably be easier to use.

Avoid the `IntInf` structure in `int-inf.sml` which implements arbitrary precision integers. A more polished implementation is part of the Basis library (see the section called *Integers* in Chapter 3).

Regular Expressions

This section describes the regular expression library in the `RegExp` directory of the SML/NJ library. This directory includes a README file which has some brief notes on using the library. This section shows some examples.

To use the library your CM files will need to include `regexp-lib.cm` (from the same place you get your `smlnj-lib.cm` file).

The Pieces of the Library

The regular expression library is designed to be very flexible. It is divided into:

- a front-end section that implements the syntax of regular expressions;
- a back-end section that implements the matching of regular expressions;
- a glue section that joins the two together.

The idea is that you can have more than one style of syntax for regular expressions e.g. Perl versus grep. The different syntaxes can be combined with different implementations of the matching algorithm. You can even feed in your own regular expressions in the internal format directly to the matching algorithms.

At the time of writing there is only one front-end which is for an Awk-like syntax. There are two back-ends. One uses back-tracking and the other compiles the regular expression to a deterministic finite-state automaton (DFA). The back-tracking matcher is described as "slow, low memory footprint, low startup cost". The DFA matcher is described as "fast, but memory-intensive and high startup cost (the cost of constructing the automaton in the first place)".

The front-end and back-end are combined together using the `RegExpFn` functor from the `Glue/regexp-fn.sml` source file. For example

```
structure RE = RegExpFn(structure P=AwkSyntax
                        structure E=BackTrackEngine)
```

The resulting structure has this signature.

```
signature REGEXP =
sig
    (* The type of a compiled regular expression. *)
    type regexp

    (* Read an external representation of a regular expression
       from a stream.
    *)
    val compile:  (char,'a) StringCvt.reader ->
                  (regexp, 'a) StringCvt.reader


    (* Read an external representation of a regular expression
       from a string.
```

```
    *)
    val compileString : string -> regexp

    (*  Scan the stream for the first occurence of the regular expression.
    *)
    val find:   regexp ->
                (char,'a) StringCvt.reader ->
                ({pos: 'a, len: int} option MatchTree.match_tree,'a)
                    StringCvt.reader

    (* Attempt to match the stream at the current position with the
       regular expression.
    *)
    val prefix: regexp ->
                (char,'a) StringCvt.reader ->
                ({pos: 'a, len: int} option MatchTree.match_tree,'a)
                    StringCvt.reader

    (* Attempt to match the stream at the current position with one
       of the external representations of regular expressions and
       trigger the corresponding action.
    *)
    val match:  (string *
                    ({pos:'a, len:int} option MatchTree.match_tree -> 'b)
                ) list
                -> (char,'a) StringCvt.reader
                -> ('b, 'a)  StringCvt.reader
end
```

Your program will first compile a regular expression using either the `compile` or `compileString` functions. You can then use one of `find`, `prefix` or `match` to match a string with the regular expression. The result of matching is a match tree. Here is the (partial) signature which defines the tree.

```
signature MATCH_TREE =
sig
    (* A match tree is used to represent the results of a nested
       grouping of regular expressions.
    *)
    datatype 'a match_tree = Match of 'a * 'a match_tree list

    (* Return the root (outermost) match in the tree. *)
    val root : 'a match_tree -> 'a
```

```
(* return the nth match in the tree; matches are labeled in
   pre-order starting at 0. Raises Subscript
*)
val nth : ('a match_tree * int) -> 'a

...
```

Each node in the tree corresponds to a regular expression in parentheses (a *group*) except the root of the tree which corresponds to the whole regular expression. Since groups can be nested you get a tree of matches. Each match tree node stores an optional pair of position and length (see the `match_tree` type in the `REGEXP` signature above). If the group was matched with part of the original string then this pair will show where. The pair is `NONE` if the group was not matched with anything e.g. if it's for an alternative that was never followed.

The matching functions are designed to work with the `StringCvt` scanning infrastructure (see the section called *Text Scanning* in Chapter 3). So for example the expression `(find regexp)` is a function that maps a character stream to a stream of match trees. To match a string you will need to combine it with the `StringCvt.scanString` function.

The `match` function takes a list of pairs of a regular expression (which will be compiled on the fly) and a function to post-process the match tree. It returns the post-processed result (of the type `'b` in the `REGEXP` signature).

All of this is very flexible but a bit verbose to use. The following sections will show some examples.

Basic Matching

This test will match the regular expression "the.(quick|slow).brown" against the string "the quick brown fox". First I build some matchers to try out.

```
structure BT  = RegExpFn(structure P=AwkSyntax
                         structure E=BackTrackEngine)

structure DFA = RegExpFn(structure P=AwkSyntax
```

```
                        structure E=BackTrackEngine)
```

Here is the function to run the matching using the BT matcher.

```
fun demo1BT msg =
let
    val regexp = BT.compileString "the.(quick|slow).brown"
in
    case StringCvt.scanString (BT.find regexp) msg of
      NONE      => print "demo1 match failed\n"
    | SOME tree => show_matches msg tree
end
```

The `scanString` function is used to apply the matcher to the message. The `show_matches` function reports the parts of the string that were matched by each group in the regular expression. Here it is.

```
(*  Show the matches n=0, ... *)
and show_matches msg tree =
let
    val last = MatchTree.num tree

    fun find n =
    (
        case MatchTree.nth(tree, n) of
          NONE => "<Unmatched>"

        | SOME {pos, len} => String.substring(msg, pos, len)
    )

    and loop n =
    (
        print(concat[Int.toString n, " => ", find n, "\n"]);
        if n >= last then () else loop(n+1)
    )
in
    loop 0
end
```

Groups are numbered by counting left-parentheses left to right from 1. Group 0 represents the entire regular expression. The `nth` function returns the match tree node for the nth group. The `show_matches` function just

iterates for increasing values of n. The last group is given by the `num` function. The output of this test is

```
Demo 1 using BT
0 => the quick brown
1 => quick
```

Matching with a Back-End

The front-end translates a regular expression to an intermediate form which is represented by the `syntax` datatype. This is defined in the following signature from `FrontEnd/syntax-sig.sml`. The `RegExpSyntax` structure implements this signature.

```
signature REGEXP_SYNTAX =
sig
    exception CannotParse
    exception CannotCompile

    structure CharSet : ORD_SET where type Key.ord_key = char

    datatype syntax =
        Group of syntax
    |   Alt of syntax list
    |   Concat of syntax list
    |   Interval of (syntax * int * int option)
    |   Option of syntax    (* == Interval(re, 0, SOME 1) *)
    |   Star of syntax      (* == Interval(re, 0, NONE) *)
    |   Plus of syntax      (* == Interval(re, 1, NONE) *)
    |   MatchSet of CharSet.set
    |   NonmatchSet of CharSet.set
    |   Char of char
    |   Begin               (* Matches beginning of stream *)
    |   End                 (* Matches end of stream *)

    val addRange : CharSet.set * char * char -> CharSet.set
    val allChars : CharSet.set
end
```

You can build regular expressions using this datatype. This intermediate form is further translated by the back-end to its own internal

representation, for example the DFA for the DFA back-end. Each back-end has its own `compile` function to do this.

The following code shows the quick brown fox example from the previous section done this way.

```
local
    structure RE = RegExpSyntax
    structure CS = RE.CharSet

    val dot = RE.NonmatchSet(CS.singleton #"\n")

    fun cvt_str s = RE.Concat(map RE.Char (explode s))
in
    fun demo2BT msg =
    let
        (* "the.(quick|slow).brown" *)
        val regexp = BackTrackEngine.compile(RE.Concat[
                        cvt_str "the",
                        dot,
                        RE.Group(RE.Alt[
                            cvt_str "quick",
                            cvt_str "slow"]),
                        dot,
                        cvt_str "brown"
                        ])
    in
        case StringCvt.scanString (BT.find regexp) msg of
          NONE      => print "demo2 match failed\n"
        | SOME tree => show_matches msg tree
    end
end
```

The dot in a regular expression usually means any character excluding the new-line character. I can achieve this with `NonmatchSet` which means all characters but the one in the set. Look at the `ORD_SET` signature for the available operations on character sets.

The `cvt_str` function converts a string to a sequence of character matchers. The syntax value is not the simplest since the `cvt_str` calls produce redundant nesting of Concats. If you were going to be doing a lot of this sort of thing it would be useful to write a normalising function that flattened

nested Concats. The Group constructor signals a group of characters to be put into the match tree. The result is the same as before.

```
Demo 2 using BT
0 => the quick brown
1 => quick
```

Other Utilities

I will only briefly mention the remaining sections of the SML/NJ utilities library.

Parsing HTML

The `HTML` directory contains a parser for HTML. It follows version 3.2 of HTML fairly strictly. It won't cope with non-standard tags and attributes the way a browser would. In a program it might be useful for reading and displaying help files or on-line documentation. Such files could be trusted to use a restricted safe form of HTML. There is a `test-parser.sml` source file which demonstrates using the parser. For debugging, there is a pretty-printer for the parsed HTML.

INet

The INet directory contains some utility functions for handling sockets. These utilities aren't written for a CML environment so I didn't use them in the web server.

Pretty-Printing

Pretty-printing means formatting a data structure such as a tree ready to print to a file. The formatting can involve indenting to show structure and styles such as a bold font or colour where possible. A pretty-printer is important in a compiler to format an expression into an error message. You might find similar uses for it in your applications.

The pretty-printer is in the `PP` source directory. The formatter is generic over different kinds of "output device". For example there is an output device for producing HTML. The `tests` directory contains some example code.

Reactive

The `Reactive` directory contains a fragment of a system for implementing a reactive programming language. The README says it all.

Unix

The `Unix` directory contains some minor functions for manipulating Unix environment variable lists and directory path lists. The environment variable functions could be useful when building the environment for a child process.

Chapter 6. Concurrency

This chapter introduces the concurrency mechanisms of the Concurrent ML library, or CML for short(see [*CML*]). The concurrency model is based around a collection of threads which communicate by sending messages (rather than sharing access to variables). CML does not use the "kernel" threads of the operating system. Instead its implementation is based on coroutines. However a timer mechanism triggers pre-emptive scheduling of the threads. The coroutine mechanism is in turn based on the idea of continuations.

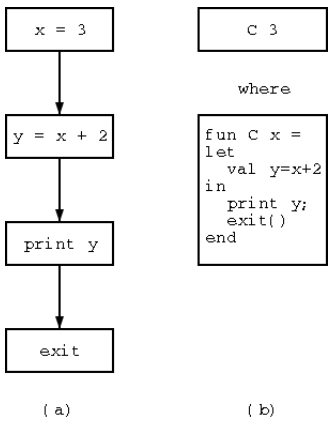
The CML library adds a collection of modules containing concurrent operations and also replaces some of the Basis and Utility library modules with thread-safe versions. There is some reference documentation provided with the library in HTML and Postscript formats. The text book on CML is [*Reppy*]. This chapter will cover the concepts in enough detail to carry on with the project in part II.

There is a bug in the version of CML distributed with SML/NJ 110.0.7. You will need to get a copy from the CML home page dated later than 14 Jan 2001 to run all of these examples successfully. The Appendix C explains how to do this.

Continuations

In the section called *Tail Recursion as Iteration* in Chapter 2 and the section called *Tail Recursion for Finite State Machines* in Chapter 2 we saw how a function call can be equivalent to and as light-weight as the `goto` of a language like C. Turning our view-point around we can represent any transfer of control as a call to some function. In the simple flow-chart of Figure 6-1(a) there are implicit transfers of control from one block to the next. Data values are also passed (implicitly), for example the value of `x` is passed to the addition in the second block.

Figure 6-1. A Simple Flow Chart



The transfer from the first to the second block can be modeled as a function call by considering everything that happens from the second block onwards as being the computation of a function. This function will be passed the value of `x` as an argument. The function is said to continue the execution of the program after the first block. This continuation function is shown in Figure 6-1(b) as the function `C`. The assignment to `x` can be reduced to a call to `C` passing the value 3 as the argument `x` of the continuation. The transfers of control inside the function `C` can be further decomposed into calls to continuation functions.

Since this function `C` represents all the rest of the execution of the program it never returns. So any call to it must be a tail call. These are the essential characteristics of a continuation: *a tail call is made to a continuation to continue the rest of the execution of the program, passing as arguments all the values that will be used by the rest of the program*. A continuation function is often passed in as an argument to a function to give greater flexibility in the choice of direction for the program.

Continuations were introduced in the study of the semantics of programming languages. A notation called Denotational Semantics was developed in the late 1960s and 1970s by Scott and Strachey among others

to describe formally the semantics of programming languages. See [*Allison*] for an introduction. Denotational semantics was based on lambda calculus which gave it a functional style. This created the problem of how to represent the control-flow of an imperative language. Continuations were invented to model control-flow in lambda calculus. They are a primitive notion that can be used to model any flow of control, including long-distance transfers such as raising exceptions. For example an `if` statement in C has a *then* and an *else* part and control is transferred to one of them. This can be modeled by having two continuations. The first contains the execution of the *then* part followed by the rest of the program. The second contains the execution of the *else* part followed by the rest of the program. The predicate of the `if` statement chooses which continuation to call.

At any point in the execution of the program we can define a *current continuation*. This is a hypothetical continuation function that represents the rest of the execution after that point in the program. If we could capture the current continuation as a real function we would obtain a snapshot of the execution of the program at that point. Some programming languages provide this feature, called *call with current continuation* or `call/cc`. Scheme for example provides this. So does SML/NJ. The `call/cc` operation captures the current continuation, from the point after the `call/cc`, and passes it as a function value to a function that you provide. Your function can do anything with the continuation including storing it for later use. If the program later calls the continuation it results in a resumption of execution from after the `call/cc`.

When you continue a program by calling a continuation function you will probably be continuing the execution of a number of called functions which have piled up on the call stack. The continued execution will include returning from called functions to functions higher up the call stack. So the contents of the call stack are an essential part of the snapshot represented by the continuation. But as an extra complication you can call a continuation more than once, just as you can call any function more than once. This results in the mind-bending possibility of rerunning parts of your program, possibly restarting in arbitrary locations in the middle of functions. You

could call a function once but it returns twice. So when `call/cc` makes its snapshot it, at least in principle, needs to save a copy of the call stack. This can be expensive. Some implementations of Scheme do exactly this.

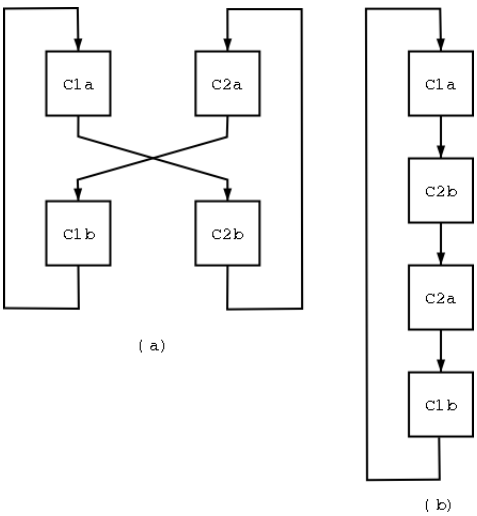
The implementation of `call/cc` in SML/NJ is much simpler. The language is implemented using continuations. The compiler identifies all continuations in the program before generating machine code for each continuation. See [*Appel2*] for the details. The call stack is maintained in the heap so that it does not need to be destroyed as functions return, as happens in the C language. So it costs nothing to save the call stack when a `call/cc` is done, since it is already in the heap. It is only necessary to retain a pointer to the top of the call stack so that the garbage collector does not take it.

With `call/cc` being almost zero cost in SML/NJ you can use it frequently for all sorts of tricks. But unless you are careful you will get code that is as spaghetti in nature as the worst Fortran or assembly language program. I won't use `call/cc` directly in this book. Instead its use will be stereotyped by CML. CML uses `call/cc` to implement switching execution between threads. The stereotyped pattern is the coroutine.

Coroutines

You are already familiar with the idea of a subroutine. Control is transferred to some separate section of code which executes to completion. Then control is returned back from whence it came. The separate section of code is subordinate to the calling code, hence the name subroutine. Coroutines redefine the relationship between two sections of code to make them more like peers. Figure 6-2(a) shows a loop with two coroutines passing control back and forth between them.

Figure 6-2. Two Coroutines



Coroutine 1 consists of two blocks of code, C1a and C1b, similarly for coroutine 2. Coroutine 1 transfers control after C1a to coroutine 2 which in this case picks up execution at the beginning of C2b and continues with C2a and then transfers control back to coroutine 1. When control is transferred to a coroutine the execution always continues at the point the coroutine was last at (except the first time it is called in which case it starts at the top of the coroutine). You might ask why not simply include the blocks inline as in Figure 6-2(b). The answer is the same as for subroutines, a coroutine might be called from multiple places or, even if called from only one place, the resulting code can be clearer.

An example of the use of coroutines is to implement the idea of *generators* for loops. Here is an example (in Perl) of iterating over a list of names.

```
foreach my $k ("tom", "dick", "harry")
{
    print "not for any $k\n";
}
```

The list could instead be produced from a function call. We can talk about a function that generates a list and a loop that iterates over or consumes the list. This Perl example consumes one value from the list for each iteration. If we had a function generating the list it would construct the complete list in memory before the loop started. This could consume a lot of memory. A more efficient way would be to have the generating function and the consuming loop run concurrently. The generator generates a value in the list and returns it to the consumer. The consumer calls back to the generator for the next value. The generator and consumer are running as coroutines.

Some languages have included generators directly as a language feature. Here is an example from the CLU language [*CLU*].

```
start_up = proc()
    outstream:stream := primary_output()
    for s:string in get_hello_world() do
        stream$putl(outstream,s)
      end
  end

get_hello_world = iter() yields(string)
    while (true) do
        yield ("Hello, World!")
      end
  end
```

In CLU generators are a special type of procedure called an iterator or `iter`. The main function `start_up()` has a `for` loop that iterates the string variable `s` over the stream of strings produced by the generator `get_hello_world()`. The generator produces an infinite stream of "hello world" messages. The `yield` statement transfers control back to the calling loop. When the calling loop completes an iteration it will transfer control back to `get_hello_world()` after the `yield` statement and the `while` loop will go around again.

Coroutines provide a more general mechanism that you can use to implement patterns like this.

In the section called *Pure FP and I/O* in Chapter 1 I talked about lazy streams. They are another example of the producer/consumer relationship.

You could implement lazy streams as coroutines. The compiler for a lazy functional language could be said to automatically convert functions to coroutines when there is a producer/consumer relationship.

Finally it is only a small step from coroutines to concurrent tasks. A set of tasks without pre-emptive scheduling is equivalent to a set of coroutines. Each task explicitly transfers control to another through a *yield* operation. If you add a timer to force the yield periodically then you have a proper pre-emptively scheduled concurrent system.

CML uses the `call/cc` operation to save the state of a running thread as a continuation. The continuations of the threads that aren't running are stored in a queue. When the current thread yields or is pre-empted a scheduler selects the next continuation from the queue and calls it to continue the thread. A timer is used to trigger a schedule of the current thread or a thread can yield when it performs some concurrent operation such as stopping to wait for a message. CML provides modified Basis library modules so that I/O can be safely preempted.

We can turn our view-point around now and use concurrent threads as a way to implement coroutines, lazy streams and any other kind of concurrent producer/consumer relationship. For example, a generator for a loop can be implemented as a thread that sends the list values as messages to a consuming thread. The generator will block until the consumer takes the next message.

The CML Model

CML provides the following concurrent features:

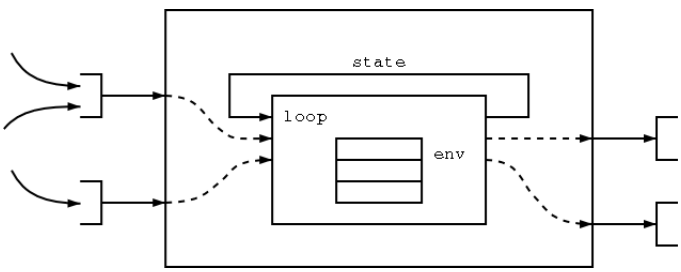
- threads;
- channels for passing messages;

- events to signal when a message has arrived, I/O has completed or a timeout has occurred;
- synchronous variables;
- mailboxes which are buffered asynchronous channels.

CML Threads

Figure 6-3 shows the typical structure of a thread in a CML program. The thread may receive messages from one or more input channels on the left. It may write messages to one or more output channels on the right. The thread may also do conventional file I/O. The body of the thread is implemented as a function. Typically this will contain a loop that runs a state machine. The function maps the pair (current state, inputs) to (next state, outputs) and then it loops. The function contains an environment which is the set of variables captured by it from surrounding scopes or passed in as arguments. These will typically supply the channels, files etc. that the thread will communicate on. The function can be written to be pure with all of its state being passed through the arguments.

Figure 6-3. A CML Thread



Threads can be created dynamically and are light-weight enough that you can structure your program with large numbers of threads.

Messages on a channel are normal values and can be of any type. However each channel has a single type. If you need to pass values of different types then you will need to either combine them into a datatype or have separate channels. Functions dealing with channels can be polymorphic over the type of the channel.

The CML model has some parallels with the object-oriented model. Originally classes and objects were introduced in the Simula language [*Simula*] which was designed for simulating (discrete event-based) real-world systems. Objects represented real-world entities. Today it is common to explain object-oriented concepts by showing how to model the real-world entities as objects and describe the differences, commonalities and relationships between objects using classes. Each object interacts with the other objects by sending and receiving messages. An object contains some private state which may be updated as a result of the messages it receives.

But real-world systems are naturally concurrent. There have been several attempts at designing concurrent object-oriented languages but this is difficult in the imperative programming paradigm because of the necessity to protect the imperative state from concurrent update and to manage the correct ordering of update operations when parts of the system are operating asynchronously. The root of the problem is that state in imperative programs is finely divided into imperative variables and spread throughout the program creating a great many points to pay attention to.

The way I use CML is to think of each thread as representing a concurrent object in the system. The objects will be coarse-grained representing major divisions of the system architecture rather than the fine-grained "everything is an object" idea that some languages push. The body of the object is usually implemented as a pure function with all of the state of the object segregated into a single state value that is passed around outside of the function. The result is a hybrid paradigm that is imperative with objects and state at the top level and is functional at the level of the implementation of the objects.

In a conventional language like Java or C++ in a multi-threaded program a piece of code may be executed by more than one thread at a time. This

creates the need for identifying critical sections which must be executed by at most one thread at a time. You could get into similar difficulties in CML if you try to have threads updating shared reference variables. Instead, following a concurrent object paradigm, you would wrap each piece of state into an object which controls access to the state. The object updates the state in response to messages from other objects. It can then be single-threaded internally with each object having its own thread. Since CML threads are light-weight it is not a problem to have large numbers of threads.

The model I'm describing here appears more explicitly in other concurrent languages as a coordination sub-language. These languages have two parts, a sequential language for manipulating the data and a coordination language to control the interaction between the concurrent objects.

An example of a coordination language is Linda[*Linda*]. Linda is independent of the data language and can be used with a variety of languages, even C. Another interesting language is COOL, the Crisp object coordination language[*CoolCrisp*]. This coordinates concurrent objects called actors. The actors typically implement finite-state machines. They communicate via asynchronous messages (rather than the synchronous messages of CML). Getting closer to functional languages, there is the new research language called Hume[*Hume*]. This has a restricted purely functional language in the Haskell mold for the data language. All imperative state is handled at the level of the coordination language. Search for "coordination language" at Google for more examples.

CML Channels

A channel is a rendezvous point between two threads that allows them to pass a value. The value passing is synchronous. The sender of the value waits for the receiver and the receiver waits for the sender.

Each channel you create has a fixed type. The type of a channel is defined in the CML structure named `CML`[1].

```
type 'a chan
```

The type variable `'a` is the place holder for the type of values passed through the channel. If you want to pass more than one type of value then you will need to either combine them in a datatype or use more than one channel.

Channels are bidirectional. Any pair of sending thread and receiving thread can use a channel. The following functions defined in the `CML` structure deal with channels. (See the `CML` structure for more channel handling functions.)

```
val channel     : unit -> 'a chan
val send        : ('a chan * 'a) -> unit
val recv        : 'a chan -> 'a
val sendEvt     : ('a chan * 'a) -> unit event
val recvEvt     : 'a chan -> 'a event
```

The `channel` function creates a new channel. The `send` and `recv` functions do just what the name says. The `sendEvt` and `recvEvt` functions return events (described in the next section). The event functions allow a thread to choose between several send or receive operations.

CML Events

An event represents some activity that will be completed at a later time. An event is treated like any other value so it can be passed around and stored. An event is said to be *enabled* when its activity is completed. For example an event might represent the reception of a message on a channel or the completion of some I/O activity.

A program can choose to wait for an event to be completed. The act of waiting is called *synchronising* and is independent of launching the activity that the event represents. The program can choose one from a collection of events to synchronise on. This is similar to the traditional `select` or `poll` system call of Unix but it is more general. An event can represent any concurrent activity such as the completion of a thread.

The type of an event is defined in the `CML` structure.

```
type 'a event
```

An event has an associated data value that is returned when the program synchronises on the event. The type variable `'a` is a place-holder for the value's type. The following functions defined in the `CML` structure handle collections of events. (See the `CML` structure for more event handling functions.)

```
val wrap       : ('a event * ('a -> 'b)) -> 'b event
val choose     : 'a event list -> 'a event
val sync       : 'a event -> 'a
val select     : 'a event list -> 'a
val guard      : (unit -> 'a event) -> 'a event
val timeOutEvt : Time.time -> unit event
val atTimeEvt  : Time.time -> unit event
```

These functions build up a representation of a network of events. The `wrap` function associates a function with an event which will process the event's value after the event is synchronised on. The `choose` function represents the choice of one event from the list of events. The choice is not actually made until a synchronisation is attempted. Then the first enabled event from the list is chosen or if several are enabled then one of them is chosen non-deterministically. A synchronisation is performed on the chosen event returning the event's value.

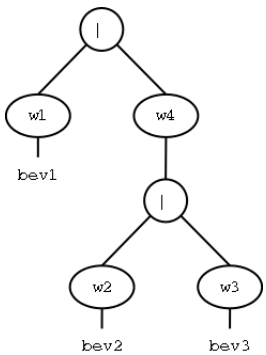
The following code illustrates the interaction of `wrap` and `choose`. The `bev` values are base events such as the reception of a message. The `w` values are wrapping functions.

```
val ev = choose [
    wrap (bev1, w1),
    wrap (choose [
        wrap (bev2, w2),
        wrap (bev3, w3)
        ], w4)
    ]
```

Figure 6-4 shows the network of events that results. The nodes labelled with "`|`" represent choices. When a synchronisation is attempted on the event `ev` then the program will wait for one of the events `bev1`, `bev2` or `bev3` to be

enabled. If `bev2` is the first to be enabled then its returned value will be run through the `w2` and `w4` functions in that order to produce the value returned by the `ev` event.

Figure 6-4. A Network of Events



The `sync` function waits for an event. The `select` function is equivalent to `choose` and then a `sync` but is more efficient.

The `guard` function associates a function with an event to be run at the time of synchronisation. The function will typically be used to make preparations for the event. This is useful in a choice of events to have preparations specific to each event.

The `timeOutEvt` function produces an event that becomes enabled after some time interval has passed. The `atTimeEvt` function is similar but it becomes enabled at a specified point in time.

Synchronous Variables

A synchronous variable is a buffer with a capacity for one value that provides for asynchronous communication between threads. A writer can put a value into a variable without waiting for a reader. There are two kinds, an

I-variable is write-once, an M-variable can be written to more than once. The writer cannot overwrite the value in an M-variable. It must wait for it to be emptied by the reader before another value can be written. Symmetrically a reader must wait for a writer to put a value into a variable. CML events are available for waiting on a variable.

I-variables are useful when you want to pass only one message between two threads. For example when replying from a remote procedure call (RPC). They are more efficient than channels for this. For a more complex example, the CML library provides an implementation of multicasting using I-variables.

M-variables are a more general-purpose primitive for building up synchronisation operations. I've only used them to implement a *mutex* to protect a critical section. The value in the M-variable can be treated as a baton which gives access to the critical section. A thread takes the baton out of the M-variable, performs the critical code and puts the baton back into the M-variable. If another thread tries to take the baton at the same time it will block because the M-variable is empty.

Synchronous variables are implemented in the `SyncVar` structure of CML. See the CML documentation for more details.

Mailboxes

A mailbox provides a buffer with unlimited capacity for asynchronous communication between threads. They are implemented in the `Mailbox` structure of CML. Since the capacity is unlimited a program using them should implement some sort of flow control.

A Counter Object

This section shows a simple example of two threads sending a message. One thread will implement a counter object that increments its value in response to commands. For the body of the object I use this `counter` function.

```
datatype Message =
        MsgIsIncr of int
    |   MsgIsStop

fun counter in_chan () =
let
    fun loop count =
    (
        case CML.recv in_chan of
          MsgIsIncr n => loop (count + n)

        | MsgIsStop =>
        (
            print(concat["Count is ", Int.toString count, "\n"])
        )
    )
in
    loop 0
end
```

The possible messages are defined by the `Message` datatype. The `counter` function contains a loop implementing a state machine. The state variable is the count with an initial value of 0. When the counter is stopped it just prints its count. The channel is passed in via the `in_chan` argument when the function is started.

The client of the object is a thread running this driver function which sends some messages and stops.

```
fun driver out_chan () =
let
in
    CML.send(out_chan, MsgIsIncr 3);
    CML.send(out_chan, MsgIsIncr ~1);
    CML.send(out_chan, MsgIsStop)
end
```

The main thread of the program is this function called `run`.

```
fun run() =
let
    val chan: Message CML.chan = CML.channel()
    val d = CML.spawn (driver chan)
    val c = CML.spawn (counter chan)
in
    CML.sync(CML.joinEvt d);
    CML.sync(CML.joinEvt c);
    ()
end
```

I first create a channel which passes `Message` values. I've used an explicit type constraint to make this clear but type inference would have figured it out if I didn't. Then I spawn two threads for the counter and its driver. The channel is passed to each thread via the curried arguments `in_chan` or `out_chan`. (Because of the `()` argument in the function definitions, the expression `(driver chan)` is a function taking `unit` as its argument, which matches the requirement of the `CML.spawn` function.) Then I wait for each thread to terminate. The `joinEvt` function returns an event that is enabled when the thread terminates. The call to `sync` performs the wait.

A thread terminates when its function returns or it explicitly calls `CML.exit`. Thread functions always return `unit` and so does the join event so there is no mechanism for a thread to return a value when it terminates unless it sends one through a channel (or does something ugly like write to a global variable).

I don't actually need to do the wait in the `run` function. If I don't then the function will return and its thread will terminate but the spawned threads will continue. The program won't terminate until all of its threads have terminated.

Here is the main function. To start the main thread I need to explicitly call `RunCML.doit` which is a (currently) undocumented function. The second argument is an optional time interval for scheduling which defaults to 20 milliseconds.

```
fun main(arg0, argv) =
```

```
let
in
    RunCML.doit(run, NONE);
    OS.Process.success
end
```

The CM file for this program is as follows. It picks up the CML package containing the `CML`, `SyncVar`, `Mailbox`, and modified I/O structures.

```
group is
    counter.sml
    /src/smlnj/current/lib/cml.cm
    /src/smlnj/current/lib/cml-lib.cm
```

You can terminate the program early by calling `RunCML.shutdown` with an exit status. This will cause the `RunCML.doit` function to return early with the status value. For example

```
RunCML.shutdown OS.Process.failure
```

In this case you would call `RunCML.shutdown` with a success status for normal exit. For example

```
fun run() =
let
    ...
in
    ...
    RunCML.shutdown OS.Process.success
end

fun main(arg0, argv) =
(
    RunCML.doit(run, NONE)
)
```

Some Tips on Using CML

It is important that you have a direct use of the `TextIO` module in your program. This ensures that the CML version of `TextIO` is used which is properly thread-safe. Defining a function like the following somewhere will do the trick.

```
fun toErr s = TextIO.output(TextIO.stdErr, s)
```

To use any of the SML/NJ library modules use `cml-lib.cm` in place of `smlnj-lib.cm` in your CM files. This will ensure that CML-compatible versions are used. This library also contains the `Multicast` and `TraceCML` structures.

You can export your program to a heap using `SMLofNJ.exportFn`. CML is not started until the call to `RunCML.doit` in the main function. You can do no CML operations before it is started except that you can create some data structures such as channels, synchronous variables and mailboxes.

Getting the Counter's Value

The next goal is to be able to return the count from a counter object. Since the object can only respond to one message at a time it will be enough to have one channel reserved for replies. The main design goal is ensuring that the channels are used correctly so that the clients of the object can't cause deadlocks.

Remember that channels are synchronous. The client will block while waiting for the counter to receive a `get` request. When the counter has received it it will send the reply on the reply channel and the client will be blocked waiting for the reply. For correctness there must be a 1 to 1 match between request and reply and the thread that accepts the reply must be the one that sent the request.

To ensure correctness the bits and pieces of the object must be hidden within a module. Here is the signature of a structure that describes the interface to a counter object.

```
signature COUNTER =
sig
    type Counter

    val new:    int -> Counter
    val incr:   Counter -> int -> unit
    val get:    Counter -> int

end
```

To a client, the counter object is represented by a proxy of type `Counter`. The proxy encapsulates the sending and receiving performed by a client. New objects are created with the `new` function which takes an initial value for the count. The `incr` and `get` functions operate on the counter.

Here are the types that implement the counter.

```
structure Counter: COUNTER =
struct

    datatype Request =
            ReqIsIncr of int
        |   ReqIsGet

    and Reply =
            ReplyIsCount of int

    and Counter = Counter of {
                    req_chan:   Request CML.chan,
                    rpl_chan:   Reply CML.chan
                    }
```

The `Request` and `Reply` types define the messages passed between the client and counter. The counter proxy is a record containing the two channels that communicate with the counter.

There is no need to retain any other handle to the counter, such as its thread. As long as there are channels that can communicate with a thread it

will be considered to be live by the garbage collector. If a value of type Counter becomes garbage then the thread that it communicates with will also become garbage and the thread will be collected and terminated. (If I want a thread to stay alive without channels then I would have to retain its thread ID somewhere).

Here is the implementation of the counter in the `new` function.

```
fun new init =
let
    val req_chan = CML.channel()
    val rpl_chan = CML.channel()

    fun counter() =
    let
        fun loop count =
        (
            case CML.recv req_chan of
              ReqIsIncr n => loop (count + n)

            | ReqIsGet =>
            (
                CML.send(rpl_chan, ReplyIsCount count);
                loop count
            )
        )
    in
        loop init
    end

    val thread = CML.spawn counter
in
    Counter
    {
        req_chan = req_chan,
        rpl_chan = rpl_chan
    }
end
```

To create a counter I create the two channels and spawn a thread to run the `counter` function. The `counter` function gets its channels and initial value from its surrounding scope. I return a value of type `Counter` to the client.

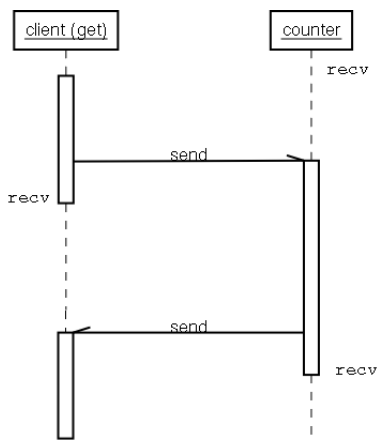
Here are the interface functions.

```
fun incr (Counter {req_chan, ...}) n =
(
    CML.send(req_chan, ReqIsIncr n)
)


fun get  (Counter {req_chan, rpl_chan}) =
(
    CML.send(req_chan, ReqIsGet);

    case CML.recv rpl_chan of
      ReplyIsCount n => n
)
```

The `incr` function just sends a message to the server to increment its value. There is no need for a reply. The `get` function stops to wait for a reply.

Figure 6-5 shows an interaction between the client and object for the `get` function. The counter has performed a `recv` and is blocked waiting for the client. When the client sends the Get message the counter runs and sends back the reply. Then it loops and blocks on a `recv` again. Meanwhile the `get` function has blocked waiting for the reply. When it gets it it returns the value to the caller.

Figure 6-5. Getting the Counter's Value



The `get` function is guaranteed to be atomic because of the synchronous nature of the `send` operation. If another thread attempts to call `incr` or `get` before a previous reply has been received then it will block at the `send` operation. The object will not receive the next message until the reply has been accepted by the client.

Getting the Value through an Event

In the previous section I have the client blocking while waiting for a reply from an interaction with the counter object. This does not mesh with the style of CML which is to separate the construction of an interaction from waiting for it to complete. What I need is to have a Get request return an event which the client can wait on. Then the client has the option of, for example, timing-out the reply or waiting for replies from multiple sources.

Events will typically be used with the CML `select` or `choose` function to decide among several interactions. When an event is chosen the program becomes commited to that choice. The event is said to represent the *commit point* in the interaction.

For the simple request-reply interaction of the counter object there are two places I can put the commit point. The simplest and most direct is to put it in the request phase of the interaction. This means that the event will be enabled when the counter is ready to accept a request. Once the event is chosen and synchronised then the program is committed to waiting for the reply. If the client were to use a time-out with this kind of event it would be timing-out waiting for the counter to become ready to accept the request, which is probably not what's intended.

Alternatively the commit point can be put in the reply phase. The event becomes enabled when the reply is received. This would allow the client to wait for a reply or for a time-out.

For the counter object, committing in the request phase will be fine since the object will reply quickly. The following example shows the counter object

with an event that commits in the request. I have added a `getEvt` function to the counter's interface. This returns an event that will deliver the counter's value when synchronised.

```
signature COUNTER =
sig
    type Counter

    val new:    int -> Counter
    val incr:   Counter -> int -> unit
    val get:    Counter -> int
    val getEvt: Counter -> int CML.event

end
```

Here is the implementation of the Get interaction.

```
fun getEvt (Counter {req_chan, rpl_chan}) =
let
    fun recv() =
    (
        case CML.recv rpl_chan of
          ReplyIsCount n => n
    )
in
    CML.wrap(CML.sendEvt(req_chan, ReqIsGet), recv)
end

fun get counter = CML.sync(getEvt counter)
```

In the `getEvt` function I use `sendEvt` to get an event that is enabled when the request has been passed to the counter. This event is wrapped with the `recv` function which will wait for the reply and return it. The original `get` function becomes a simple synchronisation on the Get event.

Because of the synchronous nature of channel communication the `ReqIsGet` message won't pass to the counter until the client has synchronised on the send event and the counter is waiting to receive. When the message is passed the client immediately goes on to wait for the reply and the counter immediately sends the reply so there is no further delay.

Getting the Value with a Time-Out

The next example shows the alternative implementation with the commit point in the reply phase. This could be used by the client with a 1 second time-out with code like this.

```
val time_out: int -> int CML.event
...
select [Counter.getEvt cnt, time_out 1]
```

As soon as the `select` starts waiting for these events I want the request to be sent to the counter. If the reply is not ready before the time-out event is enabled then the reply must be discarded. Note that for type correctness the time-out event must be able to deliver an integer the same as the counter does.

The first version of this example uses a channel to carry the reply. When the counter attempts to reply it must not become blocked while waiting for the client to accept the reply, otherwise we will lose concurrency in the program. Since sending to a channel can block, a separate thread must be spawned to deliver the reply while the counter goes on to handle the next request. So we can have more than one outstanding reply to different clients. There will have to be a different reply channel for each client.

Similarly the client should not be blocked waiting for the counter to receive the request as this may delay the start of the time-out or prevent delivery of the time-out event. So the request will be sent from a separate thread as well.

Here are the message types.

```
structure Counter: COUNTER =
struct

    datatype Request =
            ReqIsIncr of int
        |   ReqIsGet of Reply CML.chan

    and Reply =
            ReplyIsCount of int
```

```
    and Counter = Counter of {
                    req_chan:   Request CML.chan
                    }
```

The `Counter` record no longer holds a reply channel. Instead one is passed along with the `ReqIsGet` message. Here is the `new` function with the updated counter implementation. I've added a time delay into the reply for testing. If you change the delay to 2 seconds the client, shown below, will time-out.

```
fun new init =
let
    val req_chan = CML.channel()

    fun counter() =
    let
        fun loop count =
        (
            case CML.recv req_chan of
              ReqIsIncr n => loop (count + n)

            | ReqIsGet rpl_chan =>
            let
                fun reply() =
                (
                    delay 0;
                    CML.send(rpl_chan, ReplyIsCount count)
                )
            in
                CML.spawn reply;
                loop count
            end
        )
    in
        loop init
    end

    val thread = CML.spawn counter
in
    Counter
    {
        req_chan = req_chan
    }
end
```

Here is the time delay function. Don't use the `Posix.Process.sleep` function or anything similar as it will pause the entire program, not just one thread. See the section called *More on Time-Outs* for more details.

```
fun delay n = CML.sync (CML.timeOutEvt (Time.fromSeconds n))
```

Here is the updated implementation of the `getEvt` function.

```
fun getEvt (Counter {req_chan, ...}) =
let
    fun send() =
    let
        val rpl_chan = CML.channel()

        fun recv (ReplyIsCount n) = n
        fun sender() = CML.send(req_chan, ReqIsGet rpl_chan)
    in
        CML.spawn sender;
        CML.wrap(CML.recvEvt rpl_chan, recv)
    end
in
    CML.guard send
end
```

This returns an event that is constructed by the `recvEvt` function and is wrapped in both the `send` and `recv` functions. The `guard` delays the sending of the message until an attempt is made to synchronise on the event using for example `select` as shown above. Then a thread is spawned to send the Get request asynchronously and an event to represent the reception of the reply is constructed. This event is wrapped with the `recv` function to unpack the integer count in the reply. It is this receive event that is returned and waited on by the client along-side the time-out.

Finally to demonstrate the time-out I've changed the driver function.

```
fun run() =
let
    val obj = Counter.new 0

    fun time_out t = CML.wrap(
                        CML.timeOutEvt(Time.fromSeconds t),
                        fn () => ~1)
```

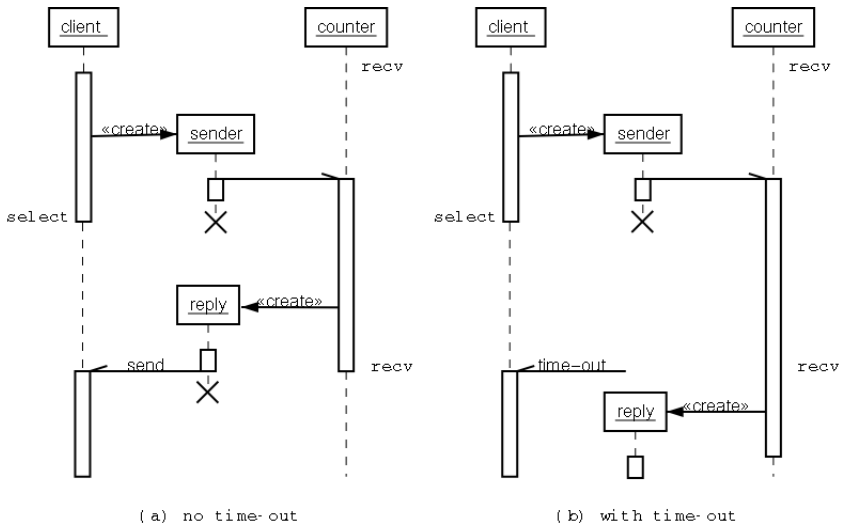
```
in
    Counter.incr obj 3;
    Counter.incr obj ~1;

    let
        val c = CML.select [Counter.getEvt obj, time_out 1]
    in
        print(concat["The counter's value is ",
                     Int.toString c, "\n"])
    end
end
```

Here I've added a `time_out` function that delivers a count of -1 if the time-out expires.

It might seem that there is a risk that one of the spawned threads will get stuck forever if either the client or the counter fails to complete the interaction. But the garbage collector can determine if a channel operation can never complete because no other threads reference the channel. So all of these spawned threads will be cleaned up properly even if there is a time-out.

Figure 6-6 shows the client getting the counter's value with and without a time-out. When there is no time-out, the `select` will choose to receive the reply. When there is a time-out the select will receive a time-out message that is triggered by the `CML.timeOutEvt` function. The reply thread running the reply function will block indefinitely. When the garbage collector collects the event from `getEvt` all references to the reply channel outside of the reply thread will disappear and the reply thread will be collected too.

Figure 6-6. Getting the Value with a Time-Out



Here is a more stream-lined version using an I-variable (see the section called *Synchronous Variables*) to return the reply. Writing into an I-variable never blocks. If there is already a value in the variable then that is an error which raises an exception. But the counter will only reply once. So the counter doesn't need to protect itself against blocking by spawning a thread for the reply.

```
structure Counter: COUNTER =
struct

    datatype Request =
            ReqIsIncr of int
        |   ReqIsGet of int SyncVar.ivar

    and Counter = Counter of {
                    req_chan:   Request CML.chan
                    }
... omitted material ...

fun counter() =
```

```
let
    fun loop count =
    (
        case CML.recv req_chan of
          ReqIsIncr n => loop (count + n)

        | ReqIsGet rpl_var =>
        (
            (* delay 2; *)
            SyncVar.iPut(rpl_var, count);
            loop count
        )
    )
in
    loop init
end
... omitted material ...

fun getEvt (Counter {req_chan, ...}) =
let
    fun send() =
    let
        val rpl_var = SyncVar.iVar()

        fun sender() = CML.send(req_chan, ReqIsGet rpl_var)
    in
        CML.spawn sender;
        SyncVar.iGetEvt rpl_var
    end
in
    CML.guard send
end
```

More on Time-Outs

There are two kinds of time-out events. The `timeOutEvt` function returns an event that will delay each time that it is synchronised on. The `atTimeEvt` function returns an event that becomes enabled at some time in the future and then is always enabled thereafter.

These differences are illustrated by the `timeouts` program. Depending on the command line argument it will exercise either of the `abs` or `delta` functions.

Here is the `delta` function.

```
fun delta() =
let
    val tmevt = CML.timeOutEvt (Time.fromSeconds 5)
in
    print "Waiting for the timeout\n";
    CML.sync tmevt;

    print "Delay\n";
    CML.sync (CML.timeOutEvt (Time.fromSeconds 1));

    print "Waiting again for the timeout\n";
    CML.sync tmevt;

    print "Done\n"
end
```

When you run the program with the `delta` command-line argument you will see that the "Waiting for the timeout" message is followed by a 5 second delay each time.

Here is the `abs` function.

```
fun abs() =
let
    val when = Time.+(Time.now(), Time.fromSeconds 5)
    val tmevt = CML.atTimeEvt when
in
    print "Waiting for the timeout\n";
    CML.sync tmevt;

    print "Delay\n";
    CML.sync (CML.timeOutEvt (Time.fromSeconds 2));

    print "Waiting again for the timeout\n";
    CML.sync tmevt;

    print "Delay again\n";
    CML.sync (CML.timeOutEvt (Time.fromSeconds 2));
```

```
    print "Waiting again for the timeout\n";
    CML.sync tmevt;

    print "Done\n"
end
```

When you run the program with the `abs` command-line argument you will see that there is a 5 second delay the first time but then each subsequent wait takes no time. The timeout event is always enabled once the target time has been reached. This will be useful in programs to produce an abort indication after a time-out.

Semaphores

In this section I demonstrate an implementation of the traditional counting semaphore for controlling access to a shared resource[2]. This example demonstrates the use of the `CML.withNack` function.

A counting semaphore has a value that can be interpreted as the number of copies of some resource that are available to be acquired. After the value falls to zero, clients that attempt to acquire a copy of the resource will be blocked until some become free.

A client will block waiting for a semaphore but may want to time-out while waiting. If it times-out then the semaphore must know that the client has given up on the semaphore. It mustn't allocate the resource to a client that has gone away. In a previous section I had a counter that sent a reply in response to a simple query. The reply was sent asynchronously from an auxillary thread. There was no way for the counter to know if the reply was received. If the client did not accept the reply then it was silently dropped. That was fine for the counter but the semaphore needs to know. This is handled with the `CML.withNack` function.

The `CML.withNack` function creates a negative acknowledgment (Nack) event that is associated with an event in a call to the `select` function in the

client. The Nack event is enabled when the `select` does not choose the associated event. The semaphore can detect the Nack event and respond.

To get things started here is the interface for a semaphore.

```
signature SEMAPHORE =
sig
    type Sema

    val new:        int -> Sema
    val acquireEvt: Sema -> unit CML.event
    val acquire:    Sema -> unit
    val value:      Sema -> int
    val release:    Sema -> unit

end
```

The `new` function creates a semaphore with an initial value which is the number of resource copies available, typically 1. The `acquire` function will acquire one copy. The `acquireEvt` version returns an event which is enabled when the resource is granted. The `value` function returns the current value of the semaphore. The `release` function returns one copy of the resource to the semaphore.

Here are the data types for the messages and objects.

```
structure Sema: SEMAPHORE =
struct
    structure SV = SyncVar

    (*  A reply channel and a nack event. *)
    type Client = (unit CML.chan * unit CML.event)

    datatype Request =
            ReqIsAcq of Client
        |   ReqIsRel
        |   ReqIsGet of int SV.ivar

    and Sema = Sema of Request CML.chan
```

The semaphore proxy only needs to contain the channel for sending requests to the semaphore. Each client must have its own reply channel since replies are delivered asynchronously and concurrently. I must use a channel, not an

I-variable, since the semaphore must get an event when the client accepts the grant. The `Client` type is used within the semaphore to represent a client with an outstanding Acquire request. It holds a copy of the reply channel and an event that will be enabled if the client abandons the request.

Here is the implementation of `acquireEvt`. I'll call the event it returns the "acquire" event.

```
fun acquireEvt (Sema req_chan) =
let
    fun sender nack_evt =
    let
        val rpl_chan = CML.channel()
    in
        CML.spawn(fn () =>
            CML.send(req_chan, ReqIsAcq (rpl_chan, nack_evt)));
        CML.recvEvt rpl_chan
    end
in
    CML.withNack sender
end

fun acquire l = CML.sync(acquireEvt l)
```

The `sender` function is a guard function that will be called when the client synchronises on the acquire event. Since the grant reply is just the `unit` value, matching `acquireEvt`, there is no need to wrap the receive event here. Here is an example of how to use `acquireEvt`.

```
CML.select[
    CML.wrap(Sema.acquireEvt sema, hold),
    time_out t
    ]
```

When the `select` starts, the `sender` function in `acquireEvt` will be called. It will be passed a newly generated event value for the Nack. (Something similar will happen within the time-out code.) The `sender` function spawns a thread which sends off the Acquire request along with the Nack event to the semaphore. Then it produces an event with `recvEvt` that is enabled when the grant is returned. Here is the implementation of the acquire message in the semaphore.

```
fun sema() =
let
    fun loop (value, pending: Client list) =
    (
        case CML.recv req_chan of
          ReqIsAcq client =>
            let         (* FIFO order *)
                val new_pending = pending @ [client]
            in
                if value <= 0
                then
                    loop (value, new_pending)
                else
                    loop (grant value new_pending)
            end
```

The semaphore's state consists of the current value and a list of clients that are waiting to acquire the semaphore. When a new Acquire request comes in I immediately append it to the list of pending clients. Even though appending to a list is expensive I decided to do it so that requests are granted in the order in which they arrive. This is an intuitively reasonable fairness condition. (If the pending queue is likely to be long then a more efficient implementation can be found in the `Queue` module of the section called *Queues and Fifos* in Chapter 5). I unconditionally add the new client to the pending list even if the semaphore is available because it makes the code neater. The list should usually be empty and appending a value to an empty list is not expensive.

Next, if the value is positive then I can grant a resource copy to a client. The `grant` function attempts the grant and returns an updated state for the semaphore.

```
(*  Look for a pending client that will accept the grant.
    Return the decremented value and the remaining pending
    clients if a client accepts the grant.
*)
and grant value [] = (value, [])        (* no pending clients *)

|   grant value ((rpl_chan, nack_evt) :: rest) =
let
    fun accepted() = (value-1, rest)
    fun nacked()   = (print "Got a nack\n"; grant value rest)
```

```
in
    CML.select [
        CML.wrap(CML.sendEvt(rpl_chan, ()), accepted),
        CML.wrap(nack_evt, nacked)
        ]
end
```

In the `grant` function I look at the first pending client. I attempt to send a grant reply to the client which would satisy the client's `select` call in `acquireEvt` if it hasn't timed-out yet. This attempt cannot block since the client must have started to wait on the grant to cause the Acquire request to be sent. So the client must either be still waiting or the Nack event will be enabled.

If the grant is accepted by the client then the `accepted` function will be run which will return the updated state for the semaphore, namely a decremented value and the rest of the pending clients.

If there is a time-out in a client then the `select` call in its `acquireEvt` will choose the time-out event. This will enable the Nack events associated with the other events that it didn't choose, namely the receive event for the grant. The Nack will propagate back to the `select` call in the `grant` function which will choose the Nack event and run the `nacked` wrapper function. This loops to try granting to one of the rest of the pending clients. I've also got a debug message in there to show what happens.

The other requests to the semaphore are simpler.

```
        | ReqIsRel => loop (grant (value+1) pending)

        | ReqIsGet rpl_var =>
        (
            SV.iPut(rpl_var, value);
            loop (value, pending)
        )
```

A Release request first increments the value. Then I call `grant` to see if the newly released copy can be granted to some pending client. The Get request just sends a copy of the value with no change of state. An I-variable is

neatest for this. Here is the implementation of the client side of these requests.

```
fun release (Sema req_chan) = CML.send(req_chan, ReqIsRel)


fun value (Sema req_chan) =
let
    val rpl_var = SV.iVar()
in
    CML.send(req_chan, ReqIsGet rpl_var);
    SV.iGet rpl_var
end
```

Because of the complexity of the semaphore I've included more testing code in the demonstration program. The first test just demonstrates the value changing from the acquire and release. The initial value is 1 so that it is a binary semaphore or *mutex*. The `check` function prints the semaphore's value.

```
fun test1() =
let
    val sema = Sema.new 1
in
    print "Test 1\n";
    check sema "1";
    Sema.acquire sema;
    check sema "1";
    Sema.release sema;
    check sema "1";
    ()
end

and check sema n =
(
    print(concat[
        "Client ", n, ": the sema value is ",
        Int.toString(Sema.value sema),
        "\n"])
)
```

The second test demonstrates contention between two clients.

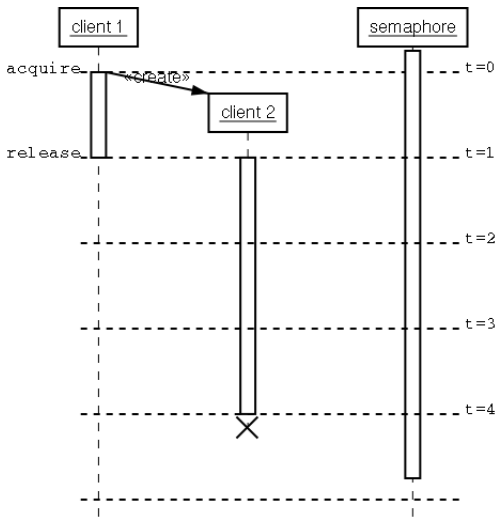
```
fun test2() =
```

```
let
    val sema = Sema.new 1
in
    print "Test 2\n";
    Sema.acquire sema;
    check sema "1";
    grab sema "2" 2;
    delay 1;
    Sema.release sema;
    check sema "1";
    delay 4;
    check sema "1";
    ()
end

and grab sema n t =
let
    fun hold() =
    (
        check sema n;
        delay 3;
        Sema.release sema;
        check sema n
    )

    fun timedout() = print(concat["Client ", n, " timed out\n"])
in
    CML.spawn(fn () =>
        CML.select[
            CML.wrap(Sema.acquireEvt sema, hold),
            CML.wrap(time_out t, timedout)
            ]
        )
end

and time_out t = CML.timeOutEvt(Time.fromSeconds t)
and delay t    = CML.sync(time_out t)
```

The thread running the `test2` function is client number 1. It acquires the semaphore and calls the `grab` function which spawns a thread for client number 2 to also try to acquire the semaphore but with a time-out. In this test there will be no time-out. Instead when client 1 releases the semaphore it will be granted to client 2 which will run the `hold` function. It will hold the semaphore for 3 seconds and release it again. Figure 6-7 shows the

timing of this interaction. The heavy lines show the times during which each client holds the semaphore. (Client 2 is spawned from client 1 and terminates at t=4).

Figure 6-7. The Timing of Semaphore Test 2



Test 3 is similar to test 2 but it delays long enough for client 2 to time-out.

```
fun test3() =
let
    val sema = Sema.new 1
in
    print "Test 3\n";
    Sema.acquire sema;
    check sema "1";
    grab sema "2" 2;
    delay 3;                (* client 2 times out *)
    Sema.release sema;      (* sema attempts to grant *)
    check sema "1"
end
```

This test produces the following debugging messages which show the time-out. You can also see that the Nack is not delivered to the semaphore until client 1 releases and the semaphore tries to grant to client 2.

```
Test 3
Client 1: the sema value is 0
Client 2 timed out
Got a nack
Client 1: the sema value is 1
```

Semaphores via Synchronous Variables

In this section I present an alternative implementation of semaphores using the M-variables that were introduced in the section called *Synchronous Variables*. This implementation is closer to the traditional implementation of languages like Java where multiple threads cooperate as peers to guarantee the safety of the critical section. This contrasts with the implementation presented in the section called *Semaphores* which relies on a central manager thread.

The simplest case is the *mutex* which protects a critical section of code so that no more than one thread can perform the code at a time. A mutex is a binary semaphore, that is one with a count of only 0 or 1. The resource is the critical section and only one copy is available for use. Here is an implementation of a mutex using an M-variable.

```
structure Mutex: MUTEX =
struct
    structure SV = SyncVar

    type Mutex = bool SV.mvar


    fun create() = SV.mVarInit true


    fun lock mutex func =
    (
```

```
        SV.mTake mutex;
        let
            val r = func()
        in
            SV.mPut(mutex, true);
            r
        end
        handle x => (
            SV.mPut(mutex, true);
            raise x
            )
    )
end
```

The M-variable either holds a value or it is empty. It doesn't matter what that value is. I've used a `bool`. The critical section is represented by the body of a function that is passed as an argument. The function doesn't call the acquire and release operations itself. This ensures that every acquire is matched by a release.

When a thread calls the `lock` function it attempts to take the value out of the mutex. If it succeeds then it can go on to run the argument function. It puts the value back into the mutex to release the lock. Other threads that call `lock` at the same time will block since the mutex is already empty. The `lock` function must be careful to release the mutex if the argument function raises an exception.

Next I would like to generalise this for counting semaphores. At first glance you might try to use an M-variable with an integer value containing the number of available copies of the resource. When the count drops to zero leave the M-variable empty so that acquirers are forced to block when they read the count. But this creates a problem for the release operation. The release mustn't block on an empty M-variable and it can't test if the M-variable is empty without a race condition or using some other mutex around the M-variable.

In the following implementation I don't let the M-variable become empty. Instead I introduce a condition variable for acquirers to block on. The design

is similar to the basic Java implementation which looks something like the following (see [*Holub*]).

```
public synchronized void acquire()
{
    while (count_ <= 0)
    {
        wait();
    }

    count_-;
}

public synchronized void release()
{
    if (count_++ == 0)
    {
        notify();
    }
}
```

The methods are synchronised to protect access to the count. If the count is zero then the calling thread is blocked and put onto a wait queue for the semaphore. The `release` method sends a notification to wake one of the waiting threads. It only does this when the count increments from zero since that is when there are acquirers waiting.

For the CML implementation I've reverted to having separate `acquire` and `release` functions, rather than the argument function of the mutex above. This is because there must be multiple sections of code in different threads using the acquired resources. I've also ignored time-outs to simplify the code. Here is the definition of the semaphore.

```
structure Sema: SEMAPHORE =
struct
    structure SV = SyncVar

    datatype Sema = Sema of {
        rsrc:   int SV.mvar,    (* count of resources avail *)
        cond:   unit CML.chan   (* signals a resource is avail *)
        }

    fun new n =
```

```
    (
        Sema {
            rsrc = SV.mVarInit(Int.max(0, n)),
            cond = CML.channel()
            }
    )
```

I use a channel to send notifications to waiting acquirers. This lets there be multiple outstanding notifications and waiting acquirers and each notification will wake one acquirer. Here is the `acquire` function.

```
fun acquire (sema as Sema {rsrc, cond}) =
let
    val n = SV.mTake rsrc
in
    if n = 0
    then
        (SV.mPut(rsrc, n); CML.recv cond; acquire sema)
    else
        SV.mPut(rsrc, n-1)
end
```

The decrement of the count is synchronised by taking it out of the M-variable. Any other thread trying to acquire will be forced to wait on the M-variable. If the count is zero then the zero is put back into the M-variable to release it and the acquirer blocks waiting for a notification on the channel. When it is notified it tries to acquire the semaphore again. Here is the `release` function.

```
fun release (Sema {rsrc, cond}) =
let
    fun notify() = ignore(CML.spawn(fn() => CML.send(cond, ())))
    val n = SV.mTake rsrc
in
    SV.mPut(rsrc, n+1);
    if n = 0 then notify() else ()
end
```

Again it takes the count out of the M-variable for synchronisation and puts the incremented value back in. If the count was zero then it sends a notification on the channel. This is done with an auxillary thread so that the

release does not block. The notifications are queued by simply leaving auxillary threads waiting for the opportunity to send.

An essential property of a correct implementation is that there be no waiting acquirers while the count is greater than zero. But proving this is rather tricky. All the different interleaving of steps over multiple acquirers and releasers must be considered. For example what happens if one or more releases happen in between the `mPut` and `recv cond` in the `acquire` function? It appears to work correctly but I'm not certain. If I allowed time-outs it would be worse. If a waiting acquirer disappeared because of a time-out there would be an excess of notifications. Would the semaphore still work correctly?

I'm more confident that the implementation of the section called *Semaphores* is correct. The protocol for dealing with the count and the waiting acquirers is implemented within a sequential manager thread. This is much easier to reason about. This is the strength of the CML paradigm. Within the boundary of a manager thread of a concurrent object the interactions between client threads are kept strictly sequential and therefore much easier to understand.

Notes

1. Read the CML reference documentation along with the following material.
2. It is based on the lock server example in section 4.2.6 of [*Reppy*].

Chapter 7. Under the Hood

This chapter discusses what goes on underneath the hood of SML/NJ and CML. First I will spend a little time discussing how memory is used in the SML/NJ run-time. Then I will examine the performance of some test programs. My goal is to give you a feel for the performance of the SML/NJ system in comparison with the C language.

Memory Management

This section describes the design of the SML/NJ heap system. It is based on a multi-generational copying garbage collector (GC).

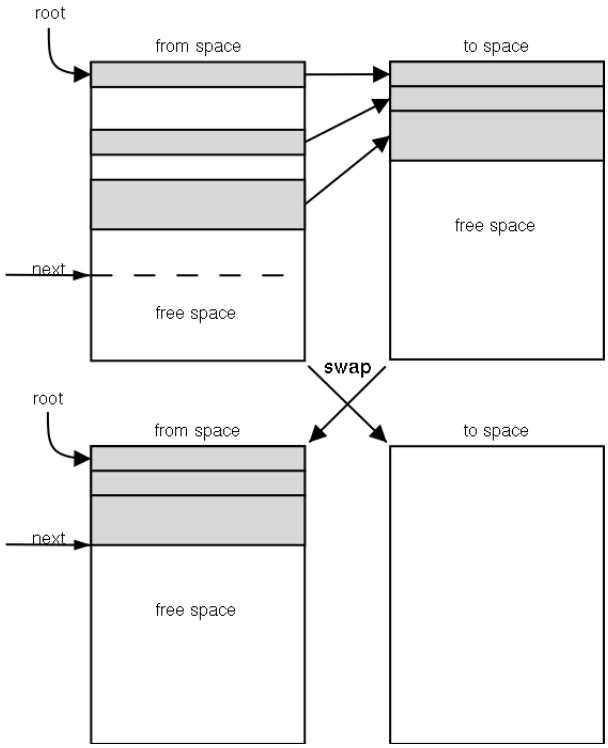
Garbage Collection Basics

A copying garbage collector works by having two memory spaces, a "from" and a "to" space. Heap objects are allocated in the "from" space until it is full. Then objects that are still live are copied to the "to" space. Then the "from" and "to" spaces are swapped. The result of the swap is a "from" space with all of the live objects and the "to" space is empty again.

Figure 7-1 illustrates these steps. Memory is allocated from the top of the "from" space advancing downwards. The arrow marked "next" is the position for new objects. The arrow is advanced down by the size of the object. When it reaches the bottom the "from" space is full.

The grey regions are live objects. A live object is any object that can be reached by following pointers starting from any of several root objects. All other objects are garbage to be removed. As the GC visits each live object it copies it to the "to" space. Then the "to" space is relabelled the "from" space and the "to" space becomes empty again. The "next" arrow is reset to the end of the copied objects ready for new objects to be allocated.

Figure 7-1. Steps in Copying Collection.



It might seem that copying the live objects would make the GC slow. But actually it's quite fast compared to other kinds of collectors. The reasons for this are:

- Only live objects are visited while tracing the pointers and only live objects are copied. The fraction of the "from" space that is live can be quite small for functional languages like SML which allocate many transient objects, perhaps around 10%. Since the dead objects are never visited the cost for deleting them is zero.

- The allocation is very fast. It only takes a few machine instructions to compare the "next" pointer with the bottom of the "from" space and advance it by the size of the object.
- After a collection the live objects have been coalesced into one memory region. This reduces the number of virtual memory pages occupied by the heap which can help with the program's performance.

When the cost of the copying is amortized over all objects that were allocated in the "from" space the cost per object is very low. It's low enough that SML/NJ does not use a separate stack for the activation records of called functions (which contain the local variables). Instead everything is allocated in the heap and the speed is competitive with stack allocation. (See [*Appel1*] for a detailed analysis of the costs). Compare this with C where you are taught that allocating objects in the heap is much slower than allocating on the stack.

Allocating activation records in the heap makes the implementation of continuations very easy and fast which in turn makes CML efficient. In effect the heap contains the stacks of each of the threads. Thread switching is fast and the GC will clean up when they terminate.

You might be worried that the copying collector wastes memory since only half of the heap space, the "from" space, is used for allocation. But no physical memory needs to be allocated to the "to" space until the copying starts and it can be removed again when the spaces are swapped. The peak amount of memory used is the size of the "from" space plus the size of the live objects (as they fill the "to" space).

Multi-Generational Garbage Collection

Even though the copying of live objects in the basic copying collector is not that slow, as explained above, it can still be improved upon. SML/NJ actually uses a multi-generational copying collector (MGGC).

The idea is that most objects are either transient and die soon or else they are long-lived. A MGGC attempts to identify the long-lived objects and copy them less often. The GC has multiple heaps called generations. A new object is allocated into the first generation. If it persists for some number of collection cycles then it is promoted into the second generation. For example there might be only one scan of the second generation for 10 scans of the first generation. This reduces the number of times that long-lived objects are copied at the cost of delaying their eventual collection and increasing the peak memory usage.

SML/NJ version 110.0.7 uses 5 generations. Each generation is a copying GC with a "from" space and a "to" space. Each older generation is scanned 5 times less often than the previous one. Persistent objects slowly migrate to the oldest generation. A "minor" collection just scans the first generation. A "major" collection scans the older generations and looks for opportunities to promote objects to the next older generation.

The SML/NJ GC has other optimisations too. Each generation is actually divided into arenas that group together objects according to their kind: records, list cells, strings and arrays. There is a separate area for "big" objects which are never copied. Currently the only big objects are those containing compiled code.

On most Unix systems the memory for the heap spaces is allocated using the `mmap` system call. The C `malloc` function continues to work separately for interfacing with the standard C library.

Run-Time Arguments for the Garbage Collector

The SML/NJ run-time takes the following arguments for the garbage collector.

@SMLalloc=<size>

This sets the size of the area where new objects are allocated, in

generation 0. The size can have a scale of K or M appended. The default is 256K bytes. Increasing this will improve the performance for programs requiring lots of memory. You will need to experiment to find the best value.

@SMLngens=<int>

This sets the number of generations. The default is 5. You cannot set more than 14. Increasing the number of generations should reduce the amount of copying at the cost of consuming more memory. You probably don't need to change this.

@SMLvmcache=<int>

When the "from" space is emptied the memory can either be returned to the operating system or kept by the run-time. This argument controls this. The default value is 2 meaning that the "from" space memory for the first 2 generations is not returned to the operating system after the copying is done. This avoids the overhead of frequently freeing and reallocating the memory. You probably don't need to change this.

Heap Object Layout

In this section I describe the layout of the different kinds of heap objects: records, list cells, strings and arrays. I won't include complete details, just the gist of it so that you can get an idea of the memory usage for SML types.

The biggest influence on the object layout is the need for the GC to be able to find all of the pointers in an object without having the details of the SML type that the object represents. This is achieved through two features of the layout:

- every object is preceded by a descriptor word that contains some type information for the whole object;

- every word in the object can be identified from a descriptor or its contents as being either data or a pointer.

The contents of strings and numeric values are known to be data just from the descriptor. In a record each field is a single 32 bit word. The pointers in the record fields are distinguished by examining the low-order 2 bits of each word. The possible combinations are:

Table 7-1. The Low-Order Bits of a Record Field.

B1	B0	Description
0	0	The field is a pointer with 32-bit alignment.
1	0	The field contains an object descriptor.
x	1	The field contains a data value in the upper 31 bits, for example the `Int.int` type.

So all data values in a record field must occupy at most 31 bits. Anything larger must be in a separate object on the heap pointed to from the first record. The first case is called an *unboxed* value and the second is called *boxed*.

The SML type `Int.int` is a 31 bit integer that is stored shifted left by 1 bit with the lower bit containing a 1 as shown in Table 7-1. You might think that it would be expensive to manipulate these integers since the machine code would have to shift the integer right when extracting it from the word and shift it left to store it again. But most of these shift operations can be avoided. No shifting is required to copy or compare the integers. Addition and subtraction only require that one of the words have its bit 0 cleared before proceeding. This is easy to arrange at no cost when one of the operands is a constant. The remaining operations including multiplication and division are relatively rare.

Since pointers are always word-aligned their low 2 bits are always zero so this fits the scheme at no extra cost.

The pointer to an object points to the first word after the descriptor.

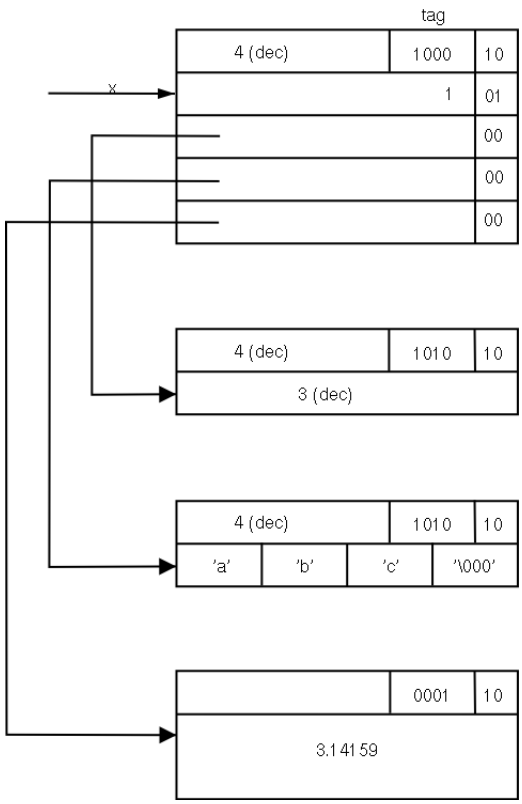
Descriptors are distinguished from all other words by their low 2 bits so that you can have a pointer into the middle of an object. The GC can always scan backwards from the pointer to find the descriptor at the top. The next 4 bits, at positions 2-5, contain a tag that indicates if the object is a record, string, array, list pair, floating point (double precision) or other kind of object.

Some objects, such as records, strings and arrays have a built-in length. This is stored in the remaining 26 bits of the descriptor word. The memory usage of a string is rounded up to a multiple of 4 bytes. This includes a terminal NUL character for compatibility with C. The length does not count the descriptor.

Figure 7-2 shows the layout of the objects corresponding to the SML this record value:

```
val x = {a = 2, b = 3:Int32.int, c = "abc", d = 3.14159}
```

Figure 7-2. The Layout of a Record.



The 32 bit integer is stored boxed as a byte vector, similar to a string. Real numbers are stored as 64 bit double precision floating point (and the length field is unused).

So the expression

```
Array.array(10, 1): Int32.int Array.array
```

will allocate 11 words for the array and 10*2 words for each boxed element for a total of 31 words. The size would be only 11 words if the element type was `Int.int`.

List cells are similar to records with two words for the head and tail and a descriptor. The empty list is represented by a zero pointer. The SML `option` type is similar. The `NONE` value is represented by a zero pointer while (`SOME` a) is represented by a record of length one containing the value. Datatypes are also like records with an extra discriminant field for the constructor. I don't have any more details on their representation.

Performance

In this section I run a few simple test programs to get a feel for the run-time cost of the elements of SML/NJ programs. In the basic performance section I look at simple loops and memory allocation. After that I look at the cost of CML operations such as message passing and thread creation.

The times that I measure are wall time because I can get finer resolution on my Linux system. The CPU timers in the SML/NJ Timer module have 10 millisecond resolution from the kernel's internal timing. As long as the system is idle while running the programs the two times should be similar enough.

These tests are run on a 1GHz Athlon system running the Linux 2.2.19 kernel. There is 256MB of PC133 memory.

Basic SML/NJ Performance

The tests I describe in this section cover some of the basic code examples to give you a feel for how fast SML/NJ runs. Remember that SML/NJ compiles direct to machine language. It's not some interpreted toy. I compare the speed to similar C code. The test programs are called `speed.sml` and

`cspeed.c`. The figures are execution times (wall time) averaged over five runs on a quiet system. The C program is compiled with Gcc 2.96 using just the basic optimisation "`cc -O`".

The first test is just a simple loop to count up to 100,000,000. I've tried two loops in SML counting up and down to see what the difference is.

```
fun countdown args =
let
    val max_cnt = int_arg args

    fun loop 0 = 0                  (* it returns something *)
    |   loop n = loop (n-1)
in
    Timing.timeIt "countdown" (fn () => ignore(loop max_cnt))
end


fun countup args =
let
    val max_cnt = int_arg args

    fun loop n =
    (
        if n = max_cnt
        then
            0
        else
            loop (n+1)
    )
in
    Timing.timeIt "countup" (fn () => ignore(loop 0))
end
```

The `countdown` function compares against the constant 0. The `countup` function compares with a variable. The C functions use a `for` loop while SML uses recursion. Table 7-2 shows the figures.

Table 7-2. Speed of the Counting Functions.

Function	SML (millisec)	C (millisec)
countdown	399	200

Function	SML (millisec)	C (millisec)
countup	449	199

You can see that SML/NJ is half the speed of C. The countup function is 10% slower for comparing with a variable compared with a constant. The C functions are the same speed in each direction. It would be interesting to study the machine code generated by SML/NJ but this is not readily accessible.

The next set of functions all count the number of lines in a text file. The number of lines is 10000 and they are all 60 characters long. This will test how SML/NJ does with character processing and I/O. Both programs read in the entire file into memory and then count the new-line characters.

In the SML program I've tried a number of different ways to count. The straight-forward C-like function is the slow index function:

```
fun count_slowix text =
let
    val len = size text

    fun loop 0 l = l
    |   loop n l =
    (
        loop (n-1) (if S.sub(text, n) = #"\n" then l+1 else l)
    )
in
    loop (len-1) 0
end
```

This indexes into the text to test each character. A faster version uses the Unsafe index function (see the section called *The Unsafe API* in Chapter 4):

```
fun count_fastix text =
let
    val len = size text

    fun loop 0 l = l
    |   loop n l =
    (
        loop (n-1) (if Unsafe.CharVector.sub(text, n) = #"\n"
```

```
                    then l+1 else l)
    )
in
    loop (len-1) 0
end
```

The remaining functions use the Substring module, just to see how much slower they are than direct indexing. The first two split the text into tokens on new-line characters using two ways of testing for a new-line. The third uses the `Substring.getc` function to step through the text.

```
fun count_tokens text =
let
    val lines = SS.tokens (fn c => c = #"\n") (SS.all text)
in
    length lines
end


(*  See if isCntrl is faster. *)
fun count_cntrl text =
let
    val lines = SS.tokens Char.isCntrl (SS.all text)
in
    length lines
end


(*  Count the characters individually using substring. *)
fun count_getc text =
let
    fun loop ss n =
    (
        case SS.getc ss of
          NONE => n
        | SOME (c, rest) =>
            loop rest (if c = #"\n" then n+1 else n)
    )
in
    loop (SS.all text) 0
end
```

The C program reads the entire file into a `malloc`ed buffer using `fread` and counts the new-lines in the usual way. Table 7-3 shows the figures. The time

to read the file is included in the `readall` function. The length entry is the time to find the length of the string which shows that it comes from a field in the string rather than counting the characters like `strlen` in C.

Table 7-3. Speed of the Line Counting Functions.

Function	SML (microsec)	C (microsec)
readall	4980	4609
length	1	
slowix	21975	
fastix	13792	1854
tokens	54856	
cntrl	61299	
getc	59050	

SML/NJ does well reading in the file. Counting the characters is woeful though. The compiler is supposed to generate in-line machine code for `Unsafe.CharVector.sub` but it still ends up 7 times slower than C. The Unsafe function is certainly faster than the normal one which has bounds checking on each call.

The Substring function use the Unsafe functions internally. I'm surprised to see that the `getc` version is slower than `tokens`.

Memory Performance

This test explores the performance of memory allocation. The program builds a linked list of integers and then frees it. For the SML/NJ program freeing consists of letting go of the list and triggering a garbage collection. Here is the test code.

```
(* lst should be garbage after this function ends *)
fun build max_cnt =
```

```
let
    fun loop 0 rslt = rslt
    |   loop n rslt = loop (n-1) (n::rslt)

    val lst = loop max_cnt []
in
    print(concat["Built a list with length ",
                 Int.toString(length lst), "\n"])
end


fun linkedlist args =
let
    val max_cnt = int_arg args

    fun run() =
    (
        build max_cnt;
        SMLofNJ.Internals.GC.doGC 0
    )
in
    run(); run();                   (* go for steady state *)
    SMLofNJ.Internals.GC.messages true;
    SMLofNJ.Internals.GC.doGC 10;   (* clear the heap *)
    print "Starting the run\n";
    Timing.timeIt "linkedlist" run;
    SMLofNJ.Internals.GC.messages false;
    ()
end
```

A separate top-level function is used for building the list to ensure that the list is truly garbage when it terminates. If it were nested within another function some compilers might retain a reference to it in the outer function's scope.

I ran the program for different list lengths to see how the performance scales. To try to ensure there is one collection I increased the heap size by adding `@SMLalloc=4096` to the run-time command line. This sets an allocation size of 4M rather than the default of 256K and the heap arenas are scaled accordingly. But I found that the speed doesn't increase for values over 1M. I always ended up with an additional major collection for lengths over 50000 which cost around 10-20 milliseconds.

Table 7-4 shows the figures for the linked list program. An estimate of the amount of time doing a major collection is included. The collection times have a 10 millisecond resolution so they are only rough.

For small list sizes SML/NJ is 3 times faster than C when allocating and freeing heap. The speed advantage largely disappears at larger sizes. The C figures are linear with the memory size. The SML/NJ figures have a hump around the 200000 level when the major collection kicks in.

Table 7-4. Speed of Linked List Building.

Length	SML (millisec)	GC (millisec)	C (millisec)
50000	5.2		15.6
100000	10.6		31.6
200000	47.2	10	64.3
500000	142.1	30	161.5
1000000	252.9	30	323.4

The bottom line is that the speed gain from faster memory allocation can compensate for the loss in raw code speed to result in execution times for SML/NJ comparable to C (or C++).

CML Channel Communication and Scheduling

This test measures how CML performs when sending messages through a channel. The test sets up a number of receiver threads all blocked on their own channel. A matching set of sender threads are started but they all first wait on a time-out event. The time-out uses `CML.atTimeEvt` to produce a single event that enables all of the senders at the same time. The receiver records the time it receives the message and the delay in sending the message. This is printed at the end of the test.

When the event becomes enabled the CML time-out code will put all of the

sender threads onto the ready-to-run queue before switching to any of the threads. This will test how the scheduler behaves when it has a large number of threads ready. When a sender thread runs and sends its message CML will immediately switch to the receiver thread. So the transmission delay will be a measure of the overhead in sending a message and switching threads. The receiver saves its record and exits which lets CML select the next sender thread to run.

Here are figures for a run with 100 threads.

```
Pair 99 receives at 1004543299.986603 after 9
Pair 98 receives at 1004543299.986618 after 2
Pair 97 receives at 1004543299.986622 after 2
Pair 96 receives at 1004543299.986626 after 2
Pair 95 receives at 1004543299.986630 after 1
Pair 94 receives at 1004543299.986633 after 2
Pair 93 receives at 1004543299.986637 after 1
Pair 92 receives at 1004543299.986640 after 3
Pair 91 receives at 1004543299.986645 after 1
Pair 90 receives at 1004543299.986648 after 2
...
Pair 9 receives at 1004543299.987025 after 3
Pair 8 receives at 1004543299.987030 after 3
Pair 7 receives at 1004543299.987035 after 2
Pair 6 receives at 1004543299.987040 after 2
Pair 5 receives at 1004543299.987045 after 2
Pair 4 receives at 1004543299.987049 after 3
Pair 3 receives at 1004543299.987055 after 2
Pair 2 receives at 1004543299.987060 after 2
Pair 1 receives at 1004543299.987064 after 3
Pair 0 receives at 1004543299.987069 after 3
...
Timing Rx 0 8
Timing Sn 0 13
Timing Rx 1 3
Timing Sn 1 4
Timing Rx 2 2
Timing Sn 2 15
Timing Rx 3 2
Timing Sn 3 3
Timing Rx 4 2
Timing Sn 4 16
...
Timing Rx 92 2
```

```
Timing Sn 92 3
Timing Rx 93 1352
Timing Sn 93 10
Timing Rx 94 4
Timing Sn 94 5
Timing Rx 95 3
Timing Sn 95 2
Timing Rx 96 2
Timing Sn 96 3
Timing Rx 97 4
Timing Sn 97 4
Timing Rx 98 3
Timing Sn 98 4
Timing Rx 99 3
Timing Sn 99 4
```

The first lines show the receiver records. The first number is the time when the message arrived and the second is the transmission delay in microseconds. The delay stays around 2 microseconds for all threads without growing. The receivers run at around 4 microsecond intervals. This includes the time to switch to a new thread, send the message and save the records. This time does not grow as the number of threads increases.

The second set of lines show the time to spawn the sender and receiver threads. The time is in the last column in microseconds. This time stays fairly stable. There are occassional spikes which may be some house-keeping inside CML.

I would rate this is as good performance.

Spawning Threads for Time-outs

In this section I examine the cost of spawning a thread and how it scales to large numbers of threads. The first test is the `thr_scaling` program and it uses time-out events. It spawns 5000 threads numbered from 5000 down to 1. The main thread creates a time-out event using `timeOutEvt` and passes it to the spawned thread which immediately waits on the event. This models an early implementation of time-outs in the Swerve server (see the section

called *Time-outs* in Chapter 8). The time-out expires well after all of the threads have been spawned. The program reports the time to spawn each thread and the order that the threads wake up.

When a thread is spawned it starts running immediately while the parent thread is blocked. The new thread then blocks on the time-out event which transfers control to the CML scheduler to choose a new thread to run. Blocked threads are placed on a time-out queue. If there is nothing else happening in the program then this queue will be examined at each time slice, typically 20 milliseconds.

The time-out queue is kept sorted in order of increasing expiry time. So as more threads are created with later time-outs they get appended to the end of the queue which takes longer and longer. But the `thr_scaling` creates time-outs with 1 second resolution and the CML scheduler uses an internal clock with a resolution of the time slice. Theis results in batches of threads on the queue with the same expiry time. The size of the batch will be determined by how many of these threads can be spawned in a time slice.

Each new member of the batch goes to the front of the queue section for the batch since its expiry time is not greater than the others. For example each member of the first batch goes to the front of the queue so this is a fast operation. The threads in the batch appear in the queue in reverse order so they will be woken in reverse order of spawning within the batch.

These quirks of the implementation help to explain the measured timing. The following data shows the time taken to perform the `CML.spawn` in microseconds and the finishing order.

```
Timing Thread 5000 18
Timing Thread 4999 17
Timing Thread 4998 5
Timing Thread 4997 3
Timing Thread 4996 16
Timing Thread 4995 3
Timing Thread 4994 3
Timing Thread 4993 4
Timing Thread 4992 4
...
Timing Thread 4374 5
```

```
Timing Thread 4373 5
Timing Thread 4372 5
Timing Thread 4371 3
Timing Thread 4370 527
Timing Thread 4369 208
Timing Thread 4368 234
Timing Thread 4367 249
...
Timing Thread 1160 9844
Timing Thread 1159 5257
Timing Thread 1158 5586
Timing Thread 1157 8607
Timing Thread 1156 5032
Timing Thread 1155 5354
Timing Thread 1154 3641
Timing Thread 1153 10322
Timing Thread 1152 3774
Timing Thread 1151 4902
...
Timing Thread 4 11012
Timing Thread 3 7851
Timing Thread 2 16275
Timing Thread 1 6906
Thread 4371 finishes
Thread 4372 finishes
Thread 4373 finishes
Thread 4374 finishes
...
Thread 4998 finishes
Thread 4999 finishes
Thread 5000 finishes
Thread 4312 finishes
Thread 4313 finishes
...
Thread 5 finishes
Thread 2 finishes
Thread 3 finishes
Thread 4 finishes
Thread 1 finishes
```

The first 630 threads spawn quickly in only a few microseconds. This would be the first batch. As the number of threads increases the time to spawn grows rapidly to over 5 milliseconds each with some much longer times up to 16 milliseconds. This is rather a long time just to set up a time-out. It resulted in poor performance in the Swerve server.

The first thread to finish is number 4371 which corresponds to the last thread in the first batch, where the spawning time jumps suddenly. This confirms the time-out queue behaviour.

Behaviour of Timeout Events

The `timeout_evt` program shows some odd behavioural differences between the events produced by the `CML.atTimeEvt` and `CML.timeOutEvt` functions.

The program spawns 1000 threads all waiting on the same time-out event. I expect that as soon as the event is enabled all of the threads will wake and terminate. If the command line argument is "time" then it will use `CML.atTimeEvt` otherwise it will use `CML.timeOutEvt`. It reports the duration of each spawn operation and the finish time for each thread.

If I use the `CML.atTimeEvt` to create the event then all of the spawn operations work in practically constant time of well under 10 microseconds. There are only a few blips where the operations take 500 microseconds or more. When the threads awake and terminate all 1000 finish in an interval of around 20 milliseconds.

If I use the `CML.timeOutEvt` function then the time for the spawn operation starts small but grows rapidly to take several hundred microseconds. When the threads awake it takes around 200 milliseconds for them all to terminate.

The reason for this behaviour stems from the implementation of the time-out queue within CML as described in the section called *Spawning Threads for Time-outs*. When using `CML.timeOutEvt`, each thread gets its own individual time-out (to the resolution of a time-slice) which is calculated at the time that the `CML.sync` on the event is attempted in the thread. Since each thread starts at slightly different times this results in many different time-out times which make the time-out queue quite long. The queue is kept in time order using an insertion sort which is rather slow. This slows down

the spawn operation since the `CML.sync` is performed before the spawn returns.

When using the `CML.atTimeEvt` there is exactly one time-out time and the time-out queue stays small.

So the lesson for good time-out performance is to keep the number of distinct expiry times and the number of waiting threads small. The final implementation of time-outs in the Swerve server goes to some lengths to achieve this. See the section called *The Abort Module* in Chapter 9.

II. The Project

Chapter 8. The Swerve Web Server

Introduction

I am now going to work through the design and implementation of a simple web server in CML, which I'll call Swerve. As well as being a good programming example it can be used as the basis for SML applications with a web interface.

The server will have the following features.

- It will implement the HTTP 1.0 protocol with the GET, HEAD and POST methods. This will be enough to support the posting of forms.
- Part of the URL hierarchy will be mapped directly to a directory hierarchy as in Apache.
- CGI-BIN scripts will be supported.
- Built-in URL handlers will be supported providing a simple servlet capability.
- There will be enough functionality to support forms via CGI.
- Multiple concurrent connections will be supported. But there will only be one server process.
- Only one language will be supported, English.

There are several similar projects around. The Fox Project has the aim of studying the application of advanced languages, in particular SML, to networks and operating system software. As a demonstration they have developed an implementation of a TCP/IP protocol stack along with a web server to show it off, see [*FoxNet*]. Everything is implemented in SML except the network device driver. The TCP/IP stack makes extensive use of functors

to plug together the various layers of the protocol stack producing an efficient implementation.

Another web server, but not in SML, is MIT's Common Lisp Hypermedia Server, see [*CLHTTP*]. It is a full-featured web server written in Common Lisp. It is used as the framework for web interfaces to some of their artificial intelligence projects as well as a few commercial projects. With more work the Swerve server could be useful for similar projects written in SML.

Yet another web server, written in Haskell and using the language's concurrent features, is described in [*HaskellServer*]. More information on using Haskell for concurrent programming can be found in [*AwkwardSquad*].

Another example of a server written in SML is the ACAP daemon which is part of the Cyrus e-mail system, see [*ACAPD*]. This implements a remote store for address books, book marks etc. according to RFC2244. It uses CML and SML/NJ so it is useful as another point of view on using CML.

The HTTP Protocol

Version 1.0 of the HTTP protocol is specified by RFC1945. You can obtain a copy from the World Wide Web Consortium, (see [*WWWC*]). (Version 1.1 of the protocol is specified in RFC2616). Also see RFC2396 for the latest general URI syntax.

The following restrictions on the protocol will be made:

- Only HTTP/1.0 will be implemented. The RFC also prescribes HTTP/0.9 but who uses that anymore?
- Only the "http" scheme will be implemented, no FTP etc.
- Only the GET, HEAD and POST methods will be implemented. GET is the basic page-fetch operation. HEAD is a variant of GET that only returnes the headers. POST will only be used to submit form data.

URL Syntax

The full monty for the URL syntax, from the RFC, is

```
URL            = ( absoluteURL | relativeURL ) [ "#" fragment ]

absoluteURL    = scheme ":" *( uchar | reserved )

relativeURL    = net_path | abs_path | rel_path

net_path       = "//" net_loc [ abs_path ]
abs_path       = "/" rel_path
rel_path       = [ path ] [ ";" params ] [ "?" query ]

path           = fsegment *( "/" segment )
fsegment       = 1*pchar
segment        = *pchar

params         = param *( ";" param )
param          = *( pchar | "/" )

scheme         = 1*( ALPHA | DIGIT | "+" | "-" | "." )
net_loc        = *( pchar | ";" | "?" )
query          = *( uchar | reserved )
fragment       = *( uchar | reserved )

pchar          = uchar | ":" | "@" | "&" | "=" | "+"
uchar          = unreserved | escape
unreserved     = ALPHA | DIGIT | safe | extra | national

escape         = "%" HEX HEX
reserved       = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+"
extra          = "!" | "*" | "'" | "(" | ")" | ","
safe           = "$" | "-" | "_" | "."
unsafe         = CTL | SP | <"> | "#" | "%" | "<" | ">"
national       = <any OCTET excluding ALPHA, DIGIT,
```

In this syntax "*()" means zero or more repetitions and "1*()" means one or more. The URL syntax allows national characters such as accented letters as long as they are 8-byte characters and include the ASCII character set. For example ISO-8859-1 "Latin 1" would be fine. This doesn't restrict the characters allowed in pages though. They are only constrained by the MIME type for the page.

Since I am only implementing the "http" scheme I will actually be implementing this syntax:

```
http_URL       = "http:" "//" host [ ":" port ] [ abs_path ]

host           = <A legal Internet host domain name
                  or IP address (in dotted-decimal form),
                  as defined by Section 2.1 of RFC 1123>

port           = *DIGIT
```

The host and scheme names are case-insensitive. If the port is empty or not given, port 80 is assumed. Only TCP connections will be used. Only absolute paths are allowed and they are case-sensitive.

The canonical form for "http" URLs is obtained by converting any uppercase alphabetic characters in the host name to their lowercase equivalent (host names are case-insensitive), eliding the [":" port] if the port is 80, and replacing an empty abs_path with "/".

Characters may be encoded by the "%" escape sequence if they are unsafe or reserved. When parsing a URL the path will be split up according to the reserved characters before escapes are interpreted. So the path `/%2Fabc/def` has `/abc` as the name of its first segment and `def` as the name of the second segment. I will reject URLs having a forward slash or a NUL character in a segment so that they can be directly mapped to file names.

HTTP Requests

Each request is done using a separate TCP connection to the server. (Version 1.1 of the protocol allows more than one request per connection which is a lot more efficient). The RFC says

> ... current practice requires that the connection be established by the client prior to each request and closed by the server after sending the response. Both clients and servers should be

> aware that either party may close the connection prematurely, due to user action, automated time-out, or program failure, and should handle such closing in a predictable fashion. In any case, the closing of the connection by either or both parties always terminates the current request, regardless of its status.

All lines in the message are supposed to be terminated with a CR-LF character pair but applications must also accept a single CR or LF character. In the body of the page the line termination will depend on the MIME type but CRLF should be used for text types.

A request message looks like:

```
Full-Request   = Request-Line
              *( General-Header
               | Request-Header
               | Entity-Header )
              CRLF
              [ Entity-Body ]

Request-Line = Method SP Request-URL SP HTTP-Version CRLF

Method       =  "GET"
               | "HEAD"
               | "POST"

General-Header = Date
               | Pragma

Request-Header = Authorization
               | From
               | If-Modified-Since
               | Referer
               | User-Agent

Entity-Header  = Allow
               | Content-Encoding
               | Content-Length
               | Content-Type
               | Expires
               | Last-Modified
               | extension-header
```

An example request line is

```
GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.0
```

After the request line come zero or more headers and then a blank line to terminate the headers. The entity body is only used to supply data for the POST method.

Each header consists of a name followed immediately by a colon (":"), a single space (SP) character, and the field value. Field names are case-insensitive. Header fields can be extended over multiple lines by preceding each extra line with at least one SP or HT (horizontal tab), though this is not recommended.

```
HTTP-header    = field-name ":" [ field-value ] CRLF

field-name     = token
field-value    = *( field-content | LWS )

field-content  = <the OCTETs making up the field-value
                 and consisting of either *TEXT or combinations
                 of token, tspecials, and quoted-string>
```

In this syntax the following definitions are used.

```
token          = 1*<any character except CTLs or tspecials>

tspecials      = "(" | ")" | "<" | ">" | "@"
                   | "," | ";" | ":" | "\" | <">
                   | "/" | "[" | "]" | "?" | "="
                   | "{" | "}" | SP | HT

TEXT           = <any OCTET except CTLs, but including LWS>

CTL            = a control character or DEL (ASCII 127)

LWS            = [CRLF] 1*( SP | HT )

quoted-string  = Any sequence of characters except double-quote and CTLs,
                   but including LWS, enclosed in double-quote characters.
                   There is no backslash quoting of characters within strings.
```

The general headers are applicable to both requests and responses. They pertain to the message itself rather than the entity being transferred. The request headers provide extra information about the request. The entity headers provide information about the entity itself. I will use them only in the response. The next sections describe the headers.

The Date Header

This provides the data and time that the message was originated. The preferred format of the date is the RFC822 format used in e-mail. For example

```
    Date: Tue, 15 Nov 1994 08:12:31 GMT
```

A well behaved server should accept all of the following date formats:

```
Sun, 06 Nov 1994 08:49:37 GMT    ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT   ; RFC 850, obsoleted by RFC 1036
Sun Nov  6 08:49:37 1994         ; ANSI C's asctime() format
```

All times are GMT (UTC). The following syntax describes all of the allowed date formats.

```
HTTP-date      = rfc1123-date | rfc850-date | asctime-date

rfc1123-date   = wkday "," SP date1 SP time SP "GMT"
rfc850-date    = weekday "," SP date2 SP time SP "GMT"
asctime-date   = wkday SP date3 SP time SP 4DIGIT

date1 = 2DIGIT SP month SP 4DIGIT          ; day month year
date2 = 2DIGIT "-" month "-" 2DIGIT        ; day-month-year
date3 = month SP ( 2DIGIT | ( SP 1DIGIT )) ; month day

time  = 2DIGIT ":" 2DIGIT ":" 2DIGIT       ; 00:00:00 - 23:59:59

wkday          = "Mon" | "Tue" | "Wed"
               | "Thu" | "Fri" | "Sat" | "Sun"

weekday        = "Monday" | "Tuesday" | "Wednesday"
               | "Thursday" | "Friday" | "Saturday" | "Sunday"
```

```
month          = "Jan" | "Feb" | "Mar" | "Apr"
               | "May" | "Jun" | "Jul" | "Aug"
               | "Sep" | "Oct" | "Nov" | "Dec"
```

The Pragma Header

This is usually "Pragma: no-cache" to tell the recipient not to cache the entity. I won't generate it.

The Authorization Header

This header provides information such as a password to access secure information. I will support basic password protection. Typically what happens is that after a request has been received, if a password is needed, the server returns a status code of 401 along with a challenge header looking like:

```
WWW-Authenticate: Basic realm="WallyWorld"
```

The client must resend the request with an Authorization header such as

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

which contains a user id and password encoded as Base64[1]. (It decodes to "Aladdin:open sesame"). The syntax for the Authorization is:

```
basic-credentials = "Basic" SP basic-cookie

basic-cookie      = <base64 encoding of userid-password,
                    except not limited to 76 char/line>

userid-password   = [ token ] ":" *TEXT
```

The client can send the Authorization with the initial request if it has already prompted the user for a password. See RFC1945 for more details for HTTP 1.0 or RFC2617 for HTTP 1.1.

The From Header

This identifies the person sending the request.

```
From: webmaster@w3.org
```

It's not normally used but I'll recognise it and pass it on.

The If-Modified-Since Header

The If-Modified-Since request-header field is used with the GET method to make it conditional: if the requested resource has not been modified since the time specified in this field, a copy of the resource will not be returned from the server; instead, a 304 (not modified) response will be returned without any Entity-Body.

An example of the field is:

```
If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT
```

I'll recognise but ignore this header.

The Referer Header

This header provides the URL from which the request originated, if appropriate. For example if a user clicks on a link in a web page then the referer URL is the URL of the page. This is sometimes used to control access to pages for example to prevent a page from being accessed unless the user has passed through a sign-on page.

An example is

```
Referer: http://www.w3.org/hypertext/DataSources/Overview.html
```

I'll recognise the header and pass it on.

The User-Agent Header

This identifies the kind of browser or whatever that generated the request. I'll recognise the header and pass it on.

The Allow Header

This is used in responses. I won't generate it. See the RFC for more details.

The Content-Encoding Header

This is used to indicate if the entity is compressed or otherwise encoded. I won't generate it in responses. An example is:

```
Content-Encoding: x-gzip
```

The Content-Length Header

This provides the size of the entity in bytes starting at the first byte after the CR-LF that terminates the header. I will always generate a content length. An example is:

```
Content-Length: 3495
```

The Content-Type Header

This provides the MIME type for the entity. An example is:

```
Content-Type: text/html
```

I will generate: text/plain, text/directory, text/html, image/jpeg, image/gif, image/png where appropriate.

The Expires Header

This is used in a response to tell the client how long to cache the document. I won't generate this.

The Last-Modified Header

This provides the date and time when the entity was last modified. I'll generate this. An example is:

```
Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT
```

Extension Headers

Any other headers are allowed as long as their syntax is valid. I'll just ignore them.

HTTP Responses

A response looks a lot like a request.

```
Full-Response  = Status-Line
                 *( General-Header
                  | Response-Header
                  | Entity-Header )
                 CRLF
                 [ Entity-Body ]

Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

Response-Header = Location
                | Server
                | WWW-Authenticate
```

The first line returns the status of the request: success, failure etc. A numeric status code is provided for programs to read. A textual version is provided for (some kind of) human readability in error messages.

The standard status codes are:

```
Status-Code    = "200"   ; OK
               | "201"   ; Created
               | "202"   ; Accepted
               | "204"   ; No Content
               | "301"   ; Moved Permanently
               | "302"   ; Moved Temporarily
               | "304"   ; Not Modified
               | "400"   ; Bad Request
               | "401"   ; Unauthorized
               | "403"   ; Forbidden
               | "404"   ; Not Found
               | "500"   ; Internal Server Error
               | "501"   ; Not Implemented
               | "502"   ; Bad Gateway
               | "503"   ; Service Unavailable
```

Full details of the status codes can be found in the RFC. I'll just describe the few that the server will use.

200 - OK

The entity follows in the Entity-Body section.

204 - No Content

Something went wrong. The Entity-Body section is omitted.

401 - Unauthorized

The client must supply a password to get the URL.

404 - Not Found

You know what this means.

500 - Internal Server Error

General cop-out.

501 - Not Implemented

I'll have a lot of this.

The response headers provide extra details for the response itself such as elaborating on the status code. They are described in the following sections. I will only use the WWW-Authenticate header.

The Location Header

This provides the location for status codes that redirect the client to some other location such as the 30x codes. An example is:

```
Location: http://www.w3.org/hypertext/WWW/NewLocation.html
```

The Server Header

This provides identification for the server e.g. its name and version. I won't be using this.

The WWW-Authenticate Header

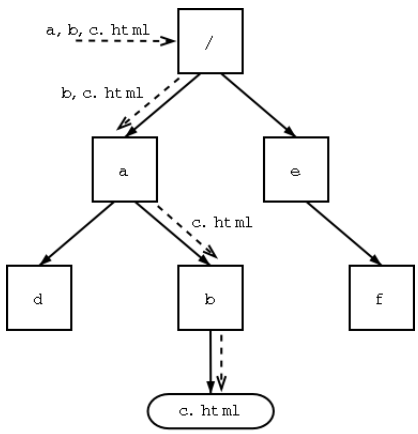
This header is returned along with a 401 status code to request the client to authenticate itself. More details can be found in the section called *The Authorization Header*.

The Resource Store

Since a URL is a universal *resource* locator I'll call the set of web pages the *resource store*. This term will also include any dynamic web page generators.

The store is a hierarchy of nodes. The URL is a path through the store from the top down. Each component of the URL path selects the next node in the store. To be nice and general and allow for different kinds of nodes each node will be passed the remainder of the URL's path and can decide for itself how to intepret this remainder. For example a node might be a CGI-BIN script that interprets query parameters. A node might also synthesise part of the store dynamically giving a different interpretation to the remainder of the path than as just a directory path on disk. Figure 8-1 shows the passage of a URL path through the store.

Figure 8-1. A URL navigating the Resource Store.



The input to the store is a list of URL path segments as described in the section called *URL Syntax*. This list is passed in a message to the root node of the store.

Each node inspects the segment list that it receives. If it is empty then the node is expected to return an entity. In the figure this is the file `c.html`.

If the list is not empty then the node inspects the first segment and chooses another node to send the remainder of the list to. If the node cannot find another node then an error will be reported.

I won't include the `~user` syntax that Apache supports.

In general a node could contain some state that is updated by the request. So each of the nodes needs to be a concurrent object in the sense of the section called *CML Threads* in Chapter 6. Different nodes can operate concurrently so that the store as a whole can handle multiple URLs concurrently. In addition a node could handle more than one request at a time by spawning a new thread for each request. The total number of threads that can be running in the store will be naturally limited by the maximum number of connections that will be allowed. A node may implement some caching for performance.

Each node will be responsible for controlling access to its part of the store. As long as all URL requests are for reading the store and do not update the store then the nodes can operate independently and concurrently. But there might be a node that is the sole representative of a resource that is updated by a request (for example a hit counter). In this case the serial behaviour of the node will synchronise access to the resource. If more than one node can alter a resource, such as a database or the file system, then they will have to cooperate through a lock manager object.

Server Configuration

This section describes how the server will be installed on a platform and how it is configured.

A typical server installation will include the following directories and files underneath the server root directory.

`swerve.cfg`	The main server configuration file is in the root directory.
`bin/`	This will contain the swerve program.
`conf/`	This will contain various configuration files such as the `mime.types` file.
`conf/icons/`	The standard Apache icons will be kept here.
`cgi-bin/`	This will contain any CGI-BIN scripts. Access to these scripts will have to be individually configured.
`htdocs/`	This is the root of the documentation tree.
`var/`	Files created by the server at run-time will be written here. This directory must not reside on an NFS-mounted file system to avoid file locking problems. Log files will be written in this directory. The space requirements are expected to be small.
`var/lock`	The lock file prevents multiple executions of the server.
`var/pid`	The pid file contains the process ID of the server.
`tmp/`	Uploaded files and data will be cached here. Lots of space may be required here. This directory must not reside on an NFS-mounted file system to avoid file locking problems.

Configuration File Syntax

A configuration file contains sections consisting of a section name followed by parameters enclosed in braces and terminated by semicolons. For example

```
Server
{
    ServerRoot = /home/swerve;
```

```
    Listen     = localhost:80;
}
```

Free use of white space will be allowed as usual. Comments begin with a # character. If the value of a parameter is not enclosed in double quotes then leading white space after the equals sign and trailing white space will be stripped.

Backslash quoting is allowed within quoted strings for the double-quote character and also `\n` for a new-line and `\t` for a tab. A backslash at the end of a line will continue a quoted string to the next line.

The names of the parameters and the sections are case-insensitive.

The Server Parameters

The following server parameters apply to the server as a whole. Where allowed, relative paths are relative to the server root unless specified otherwise.

ServerRoot

: This is the path to the server installation. It must be an absolute path. The configuration directory is expected to be at `<ServerRoot>/conf`.

VarDir

: This is the directory where run-time files are kept. It must be an absolute path and not on an NFS-mounted file system. For security this should be accessible only by the user that the server is running as. The default location is `<ServerRoot>/var`.

TmpDir

: This is the directory where temporary files are kept. It must be an absolute path and not on an NFS-mounted file system. For security this

should be accessible only by the user that the server is running as. The default location is <ServerRoot>/tmp.

Timeout

The value is the number of seconds to time-out a connection if a request or response is not completed.

MaxClients

The value is the maximum number of connections that will be allowed at a time.

MaxTmpSpace

The value is the maximum total size in bytes that temporary files in TmpDir are allowed to consume. Connections that try to upload more than this limit will be paused until space is available or they time-out.

MaxReqSize

The value is the maximum size in bytes that an entity in a request is allowed to have. These entities, such as uploaded files or form data, are saved in temporary files in TmpDir (if they are too large to be held in memory). By setting this limit you can stop one client from filling the temporary disk space.

Listen

This sets the port to listen on. This differs from Apache. The possible values are localhost:<port> or just :<port>.

ServerName

This sets the web address of the server. It is used for URLs generated by the server that refer to its own resources, for example icons. It is required.

User, Group

These set the user and group name to run as, if the server is started as root.

DirectoryIndex

This sets the name of the file in each directory that implements a node, which will be used as the index for the node. The default is `index.html`

TypesConfig

This sets the path to the `mime.types` file. This file maps from file extensions to mime types the same way as Apache. The default is `<ServerRoot>/conf/mime.types`.

The default type for an unrecognised file will be fixed at `text/plain`.

ErrorLog

This sets the path to the error log file. It must be an absolute path. The default is `<ServerRoot>/var/errors`. Unlike Apache, the log format will be fixed.

LogLevel

This sets the logging level. The possible values are: `debug`, `info`, `warn` and `error`. They are case-insensitive. The levels are ordered e.g. a level of `warn` includes all error messages as well.

The Node Parameters

The contents of the resource store is described by a series of node sections. Each legal URL path is described by a node section. The section describes how the path is implemented and what access controls apply.

An implementation of a path can be either:

a file-system directory;

When a file-system directory implements a path then optionally all subdirectories will implement extensions of the path. That is, if the URL path `/a` is implemented by the directory `/home/swerve/htdocs/a` then the subdirectory `/home/swerve/htdocs/a/b` will implement the path `/a/b`. See the WithSubDirs option below.

a CGI-BIN script;

The section will specify the path to a CGI-BIN script.

some built-in handler.

The server may have some built-in handlers which can be associated with nodes.

A node section looks like this.

```
Node /
{
    Directory = /home/swerve/htdocs;
    WithSubDirs = yes;
}
```

The URL path to the node follows the Node keyword. The root node is described by the path `/`. URL "percent escapes" are allowed in the path. Remember that URL paths are case-sensitive. The node parameters follow.

Directory = path

This specifies that the node is implemented as a disk directory. If the path is relative then it is relative to `<ServerRoot>/htdocs`.

The node will read a file named `.swerve` in its directory to look for new authorisation or option parameters. If it finds authorisation parameters then they must supply a complete valid authorisation which will replace that in the configuration section. If it finds an Options parameter then these will add to or override options in the configuration section.

BuiltIn = name

This specifies that the node is implemented by some built-in handler in the server.

Script = path

This specifies that the node is implemented by a CGI-BIN script. If the path is relative then it is relative to <`ServerRoot`>`/cgi-bin`.

Exaclty one of Directory, BuiltIn or Script must appear in each node section.

Options = [inherit | all | none] *{[+-] [FollowSymLinks | WithSubDirs]}

This specifies extra options that apply to the node. The keywords are case-insensitive.

The first keyword can be "all" to enable all options, "none" to disable all options or "inherit" to inherit options from the next higher node. Only the WithSubDirs and FollowSymLinks options can be inherited.

If this Options line appears in a `.swerve` file then the inherited options include changes made by an Options line in the configuration file's node section for the corresponding URL path.

If this parameter is omitted in the configuration file then "none" is assumed. If it is omitted in the `.swerve` file then no change is made to the node's options.

Subsequent keywords are preceded by a "+" or a "-" to enable or disable the option for this node.

The FollowSymLinks option allows symbolic links in directories to be followed (without affecting the URL path).

The WithSubDirs option allows each subdirectory to automatically implement an extension of the URL path of the node as described above for the Directory option.

AuthType = Basic

This specifies the "Basic" authentication scheme which checks user names and passwords. This parameter is required to enable authentication for access to the node.

AuthName = "Realm"

This specifies the realm name for basic authentication. A client typically maintains separate user name and password pairs for each realm.

AuthUserFile = /web/users

This specifies the path to a file that lists user names and their passwords. If the path is relative then it is relative to the server root.

The format is the same as for Apache i.e. each line is <user>:<password> except that the passwords are not encrypted. Since they aren't encrypted colons are not allowed in passwords.

There is no encryption since the Unix `crypt()` function is not available in the SML/NJ libraries and it's not important for this web server.

AuthGroupFile = /web/groups

This specifies the path to a file that lists user groups. If the path is relative then it is relative to the server root.

The file format is the same as for Apache. Each line of the group file contains a groupname followed by a colon, followed by the member usernames separated by spaces for example:

mygroup: bob joe anne

AuthAllowUsers = bob joe

This specifies a list of user names, separated by spaces, that are allowed to access the node, if they are authenticated.

AuthAllowGroups = mygroup

This specifies a list of group names, separated by spaces, that are allowed to access the node. All users in the group are allowed access if they are authenticated.

The AuthAllowUsers and AuthAllowGroups parameters together produce a list of user names. If this list is empty then all users will be allowed if their password is authenticated. To disallow all users just provide a dummy user name in AuthAllowUsers that has no password since this user can never be authenticated.

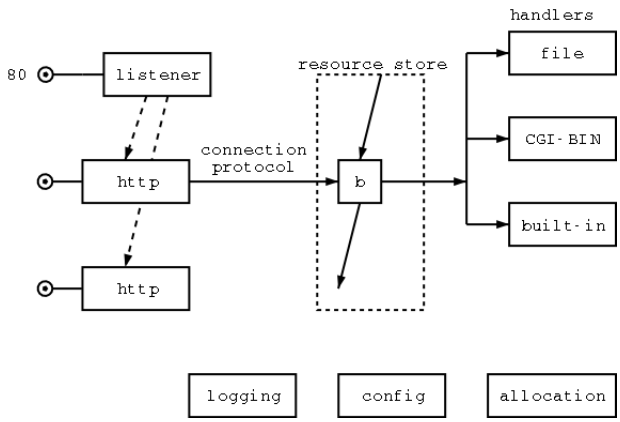
The contents of the user or group files will be read each time a request is received. This allows changes to the files to be immediately available but it will be a bit slow.

In the section called *The Resource Store* I describe how URLs are interpreted by being passed down through the tree of resource nodes. The URL must pass authentication at each node that requires it. This allows the authentication scheme at a higher node to block access to allow lower nodes.

The Architecture of the Server

The main functional blocks of the server are shown in Figure 8-2.

Figure 8-2. The Functional Blocks of the Server.



The listener object listens for incoming connections on the main port, for example port 80. For each new connection the operating system will create a new socket. The listener will spawn a new concurrent object that implements the http protocol over the socket.

The protocol object communicates with the resource store. It passes the HTTP request to the root node of the resource store. The request trickles down through the store until a node is reached that will handle it. This node has a concurrent object that performs the request. The handler transmits a response value back to the node which passes it back to the protocol object. This response value generates the body of the response on demand. For example if a file is to be returned then the file name is passed back to the protocol object which will arrange for it to be read from disk.

The messages passed between the protocol object, resource node and handler make up the connection protocol. It is implemented as SML datatypes passed over channels.

Logging is handled by a separate concurrent logging object. This ensures that logging messages from different sources don't get intermingled up. A separate logging protocol will be defined for sending logging messages.

The configuration module holds all of the configuration data for the server. Since this data is constant it is not in a concurrent object. It is just a global (singleton) module.

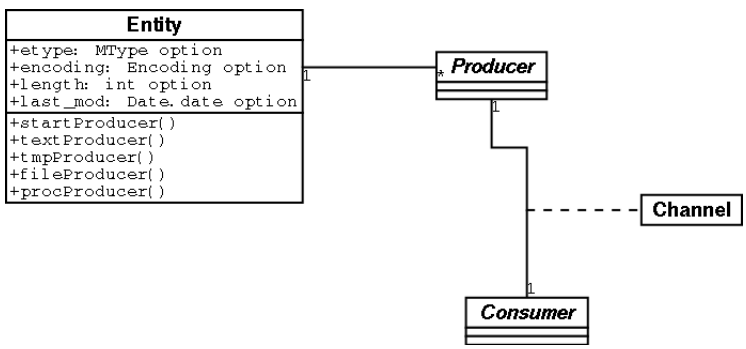
The allocation modules ration access to system resources. Limits are placed on the number of open file descriptors and the amount of temporary disk space that client connections may use. A client will be paused until the resources become available or a time-out happens.

Entities, Producers and Consumers

An entity is a central object in the server. It could be a HTML web page or image being sent to the client or a form being posted by the client. Outgoing entities can come from a disk file or be generated by a CGI script. Incoming entities need to be stored before being passed to a CGI script. It must be possible to access entities concurrently.

The producer/consumer framework supports these ways of handling entities. A producer is a thread that delivers an entity to a consumer. The delivery happens over a CML channel using the Xfer protocol. There are different kinds of producers depending on the source of the entity. Figure 8-3 shows a class diagram for these objects.

Figure 8-3. Entities, Producers and Consumers.



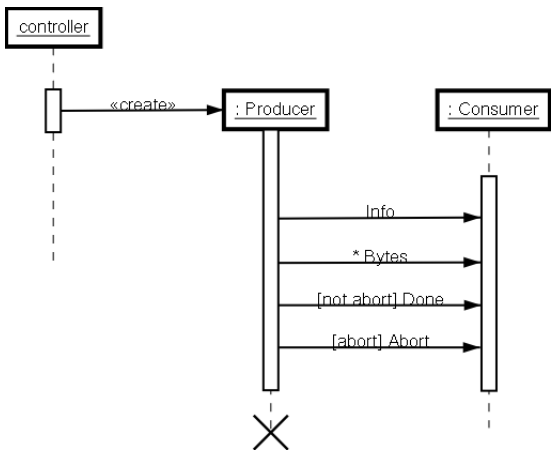
```
datatype Entity =
        Entity of {
            info:   Info,
            body:   MKProducer
            }
    |   None

and Info = Info of {
        etype:      MType option,
        encoding:   Encoding option,
        length:     int option,
        last_mod:   Date.date option
        }
```

An entity has attributes for its content type (a MIME type), encoding, length and last modification date. The encoding indicates the kind of compression used on the entity, which is not implemented at the moment. All of these attributes are optional. The length will always be taken from the size of the entity's data when available.

The body of an entity is a place holder for the data. The data could be stored in memory or in a disk file etc. If it is in a disk file then each producer opens the file concurrently. An entity can be null (`None`) with neither header nor body for the cases where HTTP requests contain no entity.

Producers and consumers are abstract classes. The framework allows any object to be a producer or consumer if it can talk the Xfer protocol over a channel. This procotol is illustrated by the sequence diagram in Figure 8-4.

Figure 8-4. The Producer-Consumer Transfer Protocol.




```
(*  A producer sends messages of this type to its consumer. *)
and XferProto =
        XferInfo  of Info               (* send this first *)
    |   XferBytes of Word8Vector.vector (* then lots of these *)
    |   XferDone                        (* then one of these *)
    |   XferAbort                       (* or else one of these *)
withtype Consumer = XferProto CML.chan
     and MKProducer = Abort.Abort -> Info -> Consumer -> CML.thread_id

(*  This creates a producer for an entity. *)
val startProducer:  Abort.Abort -> Entity -> Consumer -> CML.thread_id
```

The Info message sends the attributes of the entity. Then multiple Bytes messages send the body of the entity as chunks of bytes. The normal end of the transfer is indicated by the Done message. If the client connection is aborted, for example by a time-out, then the transfer can be aborted after the Info message by an Abort message. If an entity is null then no Info or Bytes messages are sent, only a Done. The producer thread terminates after the Done or Abort message.

An entity has a `startProducer` method that spawns a new producer thread. Most entities can spawn any number of these and run them

concurrently. An exception is for process producers. A process producer delivers its entity from the stdout of a CGI script. This can only be done exactly once. The child process will terminate as soon as the delivery as done.

In actuality in the current implementation of the server there will only be one producer for an entity. But if caching of entities in memory was implemented then it would be possible to run multiple concurrent producers.

Requests and Responses

The main objects of the protocol between a client connection and the store are the Request and Response objects. A Request contains all the information about the HTTP request from the client including details of where to return the response. The Response contains the status, headers and an entity (which may be empty).

Here is the definition of a Request.

```
datatype Request = Request of {
            method:     Method,
            url:        URL.URL,
            protocol:   string,     (* e.g. "HTTP/1.0" *)
            headers:    HTTPHeader.Header list,
            entity:     Entity.Entity,

            (*  Information from the connection. *)
            port:       int,
            client:     NetHostDB.in_addr,

            (*  The store sends its response here. *)
            rvar:       Response SyncVar.ivar,

            (*  This is the abort object from the connection. *)
            abort:      Abort.Abort
            }

and Method = GET | HEAD | POST
```

The connection information includes the client's address for CGI scripts. The port number is useful for associating resources such as temporary files with

a request. A response to the request is sent to the I-variable in the request. This allows the connection and resource store to be completely decoupled. The connection manager thread will block waiting for a response. Responses may be held up for example if there is a shortage of file descriptors or temporary file disk space.

The abort object in a request carries an indication of whether the connection has been aborted, typically because of a time-out. It provides both a flag that can be tested and a CML event that can be synchronised on. Whenever a request is to be blocked for some reason the blocking thread should also select on the abort condition to return from the wait early.

Here is the response. It just carries the status, extra headers and the entity.

```
and Response =
        Response of {
            status:     HTTPStatus.Status,
            headers:    HTTPHeader.Header list,
            entity:     Entity.Entity
            }
```

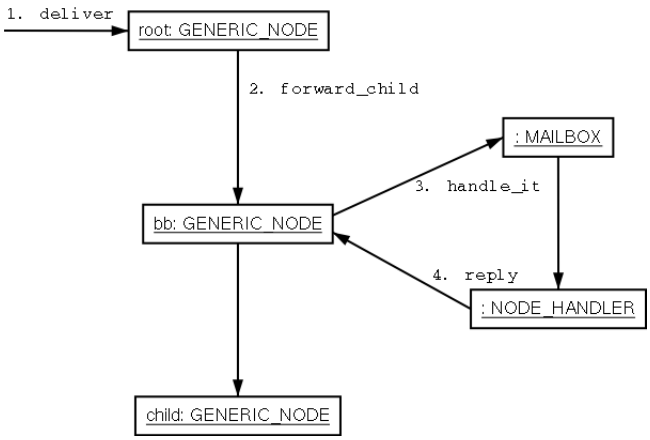
The Content-Type, Content-Encoding, Content-Length and Last-Modified headers are derived from the entity itself. The headers field provides any other headers. A CGI script can ask that a request be redirected to a different URL. This is allowed for in the Response but I haven't fully implemented it.

Resource Store Nodes

Each node in the resource store is implemented by at least two threads. The first is a backbone thread that routes the requests through the store. The second is the handler thread that runs the request. A node could be written to run multiple concurrent requests in which case the handler thread would spawn more handler threads. In the current implementation no node handles requests concurrently. The backbone node also implements common policies such as authorisation.

Figure 8-5 is a collaboration diagram that shows the interractions between the threads.

Figure 8-5. A Resource Store Node.



The backbone nodes are instances of a generic node represented in the diagram by the `GENERIC_NODE` signature. Actually a generic node is a functor which is specialised for each kind of handler. This gives the backbone node access to properties of the kind of handler which it uses to decide how to route a request.

A request enters the resource store by calling the `deliver()` function. This passes the request to the root backbone node. A backbone node will examine the URL and decide whether it is for itself or for a child node or if it can't be handled at all. If it is for a child node it will use the `forward_child()` function to pass it to the child node.

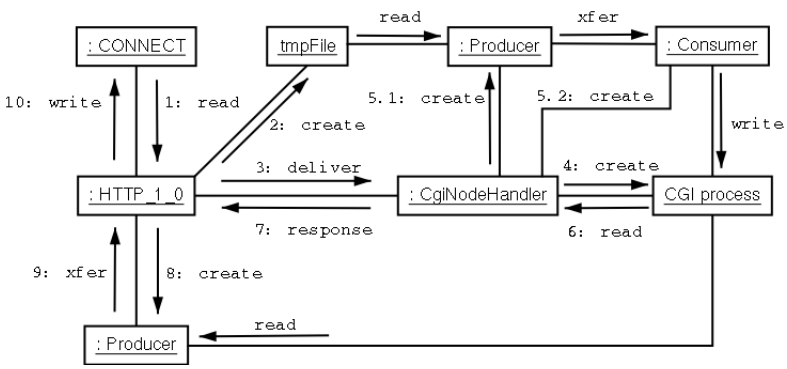
If a request is to be handled by the node it will be passed to the handler object via a mailbox. The mailbox provides unlimited buffering of requests. This frees up the backbone node for more requests. This way a slow handler node cannot prevent requests passing through to child nodes.

The handler object generates a Response value and sends it back to the backbone node. The backbone node then routes it to the client's connection object.

The Connection Protocol

The connection protocol is best illustrated by the collaboration of the HTTP protocol handler and a CGI handler. Figure 8-6 shows this while omitting the resource store nodes.

Figure 8-6. Connection-CGI Collaboration.



In step 1, the protocol handler reads a complete request from the client connection. This includes parsing the headers and reading in any entity body. It is important to read in the entire body at this point because simple clients such as Perl scripts will block waiting for the server to read the complete request before fetching any of the response. If the server tried to send a response (maybe an error response) while the client was blocked there would be some risk of a deadlock.

Bodies normally only accompany posted forms and are fairly small. Large bodies normally only come from uploaded files. If the entity body is large it

will be copied to a temporary file. Step 2 shows the creation of a temporary file. If the entity body is small it will be kept in memory and the temporary file step will be skipped. The entity object will contain the appropriate producer for the file copy or the in-memory copy.

Next, in step 3, the entire request is wrapped up and delivered to the resource store which routes it to the CGI handler. The CGI handler creates a child process in step 4 passing in the CGI environment variables. In step 5 a producer and consumer pair are created to transfer the entity body from the request to the stdin of the CGI script. The consumer is run as a separate thread to avoid a possible deadlock if the script does not read all of its input before writing some output.

Next, in step 6, the CGI handler reads the headers of the script's response. The entity body is not read at this time. Instead an entity with a special `proc` producer is created to represent the body waiting in the pipe. This entity is wrapped up with the status and headers to make a Response object which, in step 7, is sent back to the HTTP protocol handler which originated the request.

Finally, the protocol handler writes the response status and headers to the connection. In step 8 it creates a producer for the response's entity body. In step 9 it behaves as the consumer to write the body to the connection in step 10. This means that the body data is read directly out of the pipe from the CGI script and written to the network socket.

The `proc` producer deals with shutting down the CGI script process when it completes the transfer.

Time-outs

A tricky aspect of the design is dealing with abnormal events such as time-outs, broken connections etc. The connection handler starts a timer when the connection is established. If the entire request is not completed within the time-out then it must be aborted. Similarly if the connection is

broken then the server must abort further processing.

There is no way to interrupt a CML thread. Abnormal events must be delivered conventionally using a message or CML event. The threads running the request must poll for the abort condition. Ideally the abort condition is checked before every expensive operation. Everywhere a thread blocks to wait on an event, it must wait on an abort event too.

In practice it will be awkward to check comprehensively for an abort. I can allow the objects in the request processing (see Figure 8-6) to continue on for a while after the connection is aborted as long as they have no side-effects. The garbage collector will clean up the left-overs. Here are the places where I check for an abort condition.

- The HTTP protocol handler will abort waiting for a response from the resource store. It will then immediately finish with the connection.
- A receive operation from the network socket will be aborted.
- When the request waits for the allocation of some system resource it can abort.
- A producer that is transferring from a file or a child process will check for an abort condition before each data transfer. However it can't abort an I/O wait.
- Reading and writing to a CGI child process checks for abort conditions. The child process will be killed in this case.

If the client closes the connection there is no way for the server to detect this quickly. The close won't be noticed until the server attempts to write the response to the socket. At this point an I/O exception will result and an abort condition will be forced to shutdown the rest of the request handling.

System Resource Management

A robust server must not fail if it runs out of the resources it needs for the

load it is serving. This would at the least leave it open to a denial-of-service attack. The resources to worry about are

- file descriptors
- temporary file space
- the process limit

The file descriptor budget that is proportional to the number of connections is

- 1 for the socket
- 1 to write and then read a temporary file if required
- up to 4 while opening the pipes to a child process, 2 while talking to the child

This could mean 4 file descriptors per connection when running a CGI script with a large uploaded file. The file descriptor limit on my RedHat 7.1 system is 1024 so there is a possibility of running out of descriptors at a modest number of connections. (Well not really, for a small scale web server like this, but it's a good exercise). The server includes an Open File Manager that counts the number of open file descriptors. It blocks the processing of a connection until enough file descriptors are available for the connection.

Similarly a limit is placed on the total amount of disk space that temporary files can consume. A connection will be blocked until the required amount of disk space becomes available. There is also a limit on the maximum size a request can have so that a single request won't fill the disk space or block waiting for space that will never be available.

I haven't implemented a manager for the number of child processes. On my system I have a 2040 process limit. Since there is likely to be only one per connection the connection limit should be enough to prevent the server from running out of processes.

If a connection is blocked long enough due to lack of resources then it will time-out. I rely on the garbage collector to clean up after a time-out so I will need to have some sort of finalisation to close open files and remove temporary files after the collector has run. I've implemented a generic finalisation system for the server based on weak references.

Shutting Down the Server

At the moment the server is only shut down with an INTR or TERM signal. I haven't implemented any cleanup of temporary files or child processes. A better way to control the server would be to have some privileged port that allows the administrator to

- report statistics such as the current number of connections
- change the maximum number of connections
- stop the server

The proper way to shutdown the server would be to set the connection limit to zero to refuse all new connections, wait for all existing connections to close and then stop the server. I'll leave this as an exercise for the reader.

Building and Testing the Server

In the source code directory you will find a directory called `swerve` that contains the source of the server and some simple test cases. For details on the source code see the section called *The Organisation of the Code* in Chapter 9. To build the server just type `make` in this directory. This will run the SML compiler, assuming that it is in your PATH.

The resulting heap and executable files can be found in the `main` subdirectory. You will need to edit the `main/swerve` script to set the path

variables. The `libdir` variable is the absolute path where the heap file is found. This will be the main directory until you install the server somewhere else. The `smlbin` variable is the absolute path where you have installed the SML/NJ binaries.

There is a server configuration in the `tests/tree_norm` subdirectory. This contains some test cases for manual testing of the server. (The "norm" refers to testing the normal behaviour of the server). The configuration file is `norm.cfg`. Before you can run the server you must edit the `ServerRoot` parameter to be the absolute path to the `tree_norm` directory. You may want to change the port number too.

The test tree implements the notable URL paths shown in Table 8-1.

Table 8-1. The Notable Norm URLs.

Path	Purpose
/	This returns `index.html` in the `htdocs` directory.
/secure	This returns `index.html` in the `htdocs/secure` directory. This is for testing user authorisation.
/sub	This returns a list of the files in `htdocs/sub` directory. Selecting page.html tests some image handling on a complex page.
/hw	This runs a built-in handler that just returns a "hello world" message.
/sleep?3	This runs the "sleep" built-in handler that delays for the number of seconds given in the query. This is used to test simultaneous requests to the same handler.
/icons	The fancy indexing of directories refers to icons with this path. The icons are in the `kit/icons` directory.
/tmp	This path points to the `/tmp` directory on your system. On mine, having run KDE and sometimes Gnome, there are lots of weird files to exercise the fancy indexing of the server.

Path	Purpose
/cgi/env	This path runs the `cgi-bin/printenv` shell script. This is the traditional example CGI script that prints all of the environment variables that the server passes to CGI scripts.
/cgi/test-post	This runs the `cgi-bin/test-post` shell script. This is similar to `printenv` except that it echoes stdin so that posted form data can be seen.
/cgi/form-handler	This runs the `cgi-bin/form-handler` Perl script. This script runs a simple form that asks for your name and echoes it.
/cgi/line-counter	This runs the `cgi-bin/line-count` Perl script. This is used to test the uploading of a text file. The script returns the number of lines in the file.

To start the server go to the `tests/tree_norm` directory and use the `run` script. The configuration file sets the log level to Debug. The debug messages will appear in the `var/errors` log file. There are enough for you to trace the operation of the server. Note that the file continues to grow over successive runs of the server unless you delete it.

After you stop the server you should check that the `tmp` directory is empty and the `var` directory only contains the log file.

The test plan so far has been casual. Most of the tests consist of just poking at the links. The testing of multiple requests and authorisation is more involved. The following sections describe the test plan. I've tested it with both Netscape 4.76 and Konqueror 2.1.1.

Basic Testing

Try out each of the following steps from the main menu, in any order.

1. Go the main page at `http://localhost:2020/` (or whatever port you have chosen). You should see "Welcome to the Swerve Web Server" along with a menu.
2. Click on "hello world". You should see a single line of text saying "hello world". This uses the `text/plain` MIME type.
3. Click on "5 second sleep". The browser should twiddle its thumbs for 5 seconds and then you see "Slept for 5 seconds". This is more useful for multiple request testing below.
4. Click on "sub-directory". This shows a fancy index of the `htdocs/sub` directory. There should be a folder icon with the `empty` directory. Click on it to see the absence of contents. Clicking on the "Up to higher level directory" will take you back.

   Examine the README and image files. Clicking on `page.html` will show a page containing all of the image files.

5. Click on "/tmp". This will show an index of your /tmp directory. Mine contains various weird files such as sockets for KDE and X11. For example clicking on the socket in `.X11-unix/X0` will result in a "Not Found" error since the file is inaccessible. You can probably find a file that is not readable by you. Clicking on it should result in the "Access Denied" error.
6. Click on "printenv". This should return a list of all of the CGI variables passed to the script. The `HTTP_USER_AGENT` variable will describe your browser. The `SERVER_SOFTWARE` variable describes the version of the server. Your `PATH` and `SHELL` should also appear. If you click on "printenv with a query" you will see that the `QUERY_STRING` variable appears. It should contain `name=fred%20flintstone&page=3`.
7. Click on "simple form". This returns the `testform.html` page. This contains a simple form that requests you to enter your name. If you enter Fred Flintstone and click Send the result should show that stdin

contained the string
`given=Flintstone&family=Fred&Submit=Send`.

8. Click on "real CGI form". This form is generated on-the-fly by the `form-handler` Perl CGI script. When you send your name the page is updated with the name.
9. Click on "file line counter". This shows a form that invites you to enter a file name. This should be the path of a text file that is around 100KB long. This is plenty large enough to ensure that the uploaded file is saved to disk in the `tmp` directory. You can check this in the debug messages in the `var/errors` file. Look for the line with "TmpFile allocate file" followed by the path to the temporary file. Check that the reported length of the file is correct.

Congratulations. It basically works.

Testing Multiple Requests

This testing is fairly slight. The first part relies on the browser using multiple connections to load the `page.html` file in the sub-directory of the section called *Basic Testing*. Both Netscape and Konqueror will open several simultaneous connections to the server to load all of the images.

Stop the server and edit the `norm.cfg` file to set MaxClients to 1. Restart the server and follow the sub-directory link from the main menu. You should see problems such as missing icons. If you click on the `page.html` file then images are broken. Click on Reload a few times. Two of the images are missing. If you study the log file it will not show any connection requesting the missing images. This is because the requests are going to extra connections which are being refused by the server. If you increase MaxClients to 2 then only one of the images will be missing. At 3, all of the images reappear.

This test has demonstrated that the server rejects connections according to the MaxClients parameter. If you study the log file you will see multiple

simultaneous connections. Using Konqueror to reload the image page I get this pattern of Info messages:

```
2001 Sep 19 05:31:26 Info: New connection from 127.0.0.1:2525
2001 Sep 19 05:31:26 Info: New connection from 127.0.0.1:2526
2001 Sep 19 05:31:26 Info: New connection from 127.0.0.1:2527
2001 Sep 19 05:31:26 Info: End of connection from 127.0.0.1:2525
2001 Sep 19 05:31:26 Info: End of connection from 127.0.0.1:2527
2001 Sep 19 05:31:26 Info: End of connection from 127.0.0.1:2526
2001 Sep 19 05:31:26 Info: New connection from 127.0.0.1:2528
2001 Sep 19 05:31:26 Info: End of connection from 127.0.0.1:2528
```

This shows three simultaneous connections followed by a separate one for the background image.

A different concurrent path through the server can be tested using the `sleep` built-in handler. For this you will need two browser windows side-by-side showing the main menu. Click on 5 second sleep in both browser windows rapidly one after the other. The first window will return the "Slept for 5 seconds" message after 5 seconds. The second will return it after 10 seconds. This is because the handler is single-threaded. The second request waits in a queue for the first one to finish. Then it starts its own 5 second delay. This test demonstrates multiple simultaneous requests routed through the resource store to the same handler and handled in turn.

Testing Authorisation

In the `tests/tree_norm/conf` directory you will find a password file, `admin.pwd`, and a group file, `admin.grp`, for testing the authorisation. (See the section called *The Node Parameters*). Here is the password file.

```
fred:rocks
wilma:pebbles
barney:bowling
betty:mink
```

Here is the group file.

```
Rubble: barney betty
```

These files are used by the authorisation configuration in the `htdocs/secure/.swerve` file. Here is the file.

```
#   This allows fred directly and barney via the group.
#   Wilma should be barred.
AuthType        =       Basic;
AuthName        =       "admin";
AuthUserFile    =       conf/admin.pwd;
AuthGroupFile   =       conf/admin.grp;
AuthAllowUsers  =       fred;
AuthAllowGroups =       Rubble;
```

This allows users fred, barney and betty.

From the main test page (see the section called *Basic Testing*) click on "secure admin". You will be prompted for a user name and password. You should be able to demonstrate that you cannot gain access for users `fred`, `barney` and `betty` except with the correct password and user `wilma` cannot gain access at all. Since your browser probably remembers the user name and password for the realm (from the `AuthName` parameter) you will need to stop and restart your browser after each successful try.

Testing the Performance

Now for the big question. How does it perform? Not bad actually. To test it I used the `httperf` program which is available from Hewlett-Packard Research Labs[2].

The `httperf` program can generate a load for the server at a fixed number of connections per second. It reports the number of connections per second that the server has actually achieved along with some statistics such as the number of concurrent connections, milliseconds per connection. The program must be run on a different machine to the server to get an accurate measure of its performance. This is because it consumes a large amount of CPU time

itself in order to keep its timing accurate. But for these tests I've run it on the same machine as server.

These tests have been run on an AMD Athlon 1GHz processor with PC133 RAM. The kernel is Linux 2.2.19. The machine was idle while running the tests. The performance is compared with a standard installation of Apache 1.3.19. Both servers have logging disabled so that disk writes don't slow them down.

These performance figures were made after the improvements from profiling described in the section called *Profiling the Server*.

The tests fetch the index page of the `tree_norm` directory. This totals 957 bytes for Swerve and 1092 bytes for Apache.

The first test just tests the sequential performance. The client makes connections one after the other as fast as possible and the number of connections per second achieved is reported. The results are shown in Table 8-2.

Table 8-2. Sequential Server Performance

Server	Conn/sec
Swerve	50
Apache	680

The speed of Swerve here is terrible. But notice how it is managing exactly 20 milliseconds per connection. The default time slice for the threading in CML is 20 milliseconds. This points to a bug either in the server or the CML implementation that needs further investigation.

The next test has the client generating new connections at a fixed rate for a total of 4000 connections. If the server can keep up then it will serve them all and the number of concurrent connections it handles will be low. If it can't keep up then the number of concurrent connections will rise until it hits a limit which is 1012 connections on my machine. This limit comes from

the maximum number of sockets that the client can open at one time. If this limit is reached then the performance figure can be ignored and the server be deemed to be completely swamped.

The Swerve configuration has a Timeout of 10 seconds and MaxClients of 1100. The LogLevel is Warn so that no connection messages are written to the log file. I wait at least 10 seconds between tests to let the time-outs in the Swerve server complete. This starts each run from a clean state.

Table 8-3 shows the figures (connections/second) for Swerve. When the server starts falling behind I run multiple tests to get an idea of how variable the performance is. The `httperf` command line used is

```
httperf -timeout=15   -client=0/1       -server=brahms -port=2020
    -uri=/index.html  -http-version=1.0 -rate=350
    -send-buffer=4096 -recv-buffer=16384
    -num-conns=4000   -num-calls=1
```

Table 8-3. Concurrent Swerve Performance

Offered Rate	Actual Rate	Max Connections
100	100	8
130	130	11
150	150	28
170	170	14
190	186	21
190	181	19
190	175	16
190	170	18
200	195	20
200	191	19
200	192	30
200	189	21

Offered Rate	Actual Rate	Max Connections
210	197	21
210	198	24
210	195	22
210	193	26
220	212	39
240	206	33
240	203	32
240	170	29
240	200	29
260	160	28
260	201	64
260	197	30
260	138	36
280	206	41
280	202	31
280	219	35
280	202	57
300	215	44
300	216	91
300	228	41
300	229	42
320	182	52
320	195	52
320	173	33
320	230	34
350	123	906

Offered Rate	Actual Rate	Max Connections
350	189	96
350	238	40
350	154	47

You can see that the throughput increases linearly up to 190 conn/sec and then starts to falter. As the load increases it peaks at around an average of 210 connections per second. At a load of 350 conn/sec, connections were starting to time-out and the server was definitely overloaded.

Table 8-4 shows the figures for Apache.

Table 8-4. Concurrent Apache Performance

Offered Rate	Actual Rate	Max Connections
100	100	13
130	130	7
150	150	8
170	170	10
190	190	12
200	200	145
200	200	133
210	210	33
210	210	123
220	220	76
240	240	165
260	260	178
280	280	199
300	190	1012

Offered Rate	Actual Rate	Max Connections
300	300	132
300	245	142
300	300	189
320	215	760
320	202	1012
320	258	547
320	210	760
350	220	1012
350	277	629
350	132	1012
350	243	1012

With Apache, the through-put increases linearly up to 300 conn/sec and then starts to falter. At higher loads the throughput struggles to get up to around 270 conn/sec at best and at worst many cases of complete overload.

Comparing the two servers I can reasonably claim that the performance of Swerve is around 2/3 of Apache. That's not bad for a home-grown server written in SML.

Profiling the Server

I did some investigation of the internal timing of the Swerve server to see what improvements could be made. The performance figures in the previous section were obtained after the improvements from profiling.

I've had performance problems from the handling of time-outs. I need to have a flag that is set on a time-out and that can be tested. (See the section called *Time-outs* for more details). In an earlier implementation I created a new thread for each connection to wait for the time-out. The thread then set the

flag. The trouble with this was that the time-out thread would hang around in the server after the connection was closed, until the time-out expired. This would result in thousands of threads in the server which clogged the CML run-time. The time taken to spawn a new thread rose to over 15 milliseconds. The current implementation, described in the section called *The Abort Module* in Chapter 9, only uses one thread and is much faster.

I've left some timing code in the server which can be enabled with the `-T Timing` command line option and a log level of Debug. The timing code uses the `MyProfile.timeIt` function to measure the wall-time execution of a function, in microseconds. (See the section called *The MyProfile Module* in Chapter 9). Here are some typical figures for the fetching of the index page of the `tree_norm` test suite. (The page has been fetched several times to get it into the Linux cache).

```
Timing abort request 18
Timing abort create 47
Timing Listener setupConn 67
Timing HTTP_1_0 get 618
Timing GenNode request 165
Timing HTTP_1_0 stream_entity 641
Timing HTTP_1_0 response 764
Timing HTTP_1_0 to_store 959
Timing Listener talk 1586
Timing Listener close 53
Timing Listener release 11
Timing Listener run 1733
Timing Listener died 74
Timing Listener connection 2263
```

The measurement points are described in Table 8-5. You should study the Chapter 9 for more information on what is happening in the server.

What the numbers tell me is that the server can process a request in 2.2 milliseconds and so should be able to handle 450 requests per second. But now if I run the server with 120 requests/second to get 3 or more concurrent connections I get numbers like these:

```
Timing abort request 17
Timing abort request 7
Timing abort create 184
```

```
Timing Listener setupConn 201
Timing HTTP_1_0 get 236
Timing abort request 7
Timing abort create 557
Timing Listener setupConn 564
Timing HTTP_1_0 get 150
Timing GenNode request 460
Timing abort create 465
Timing Listener setupConn 474
Timing HTTP_1_0 get 149
Timing GenNode request 226
Timing GenNode request 7
Timing HTTP_1_0 stream_entity 1890
Timing HTTP_1_0 response 2003
Timing HTTP_1_0 to_store 2495
Timing Listener talk 2740
Timing Listener close 66
Timing Listener release 39
Timing Listener run 3062
Timing HTTP_1_0 stream_entity 1695
Timing HTTP_1_0 response 1759
Timing HTTP_1_0 to_store 2477
Timing Listener talk 2633
Timing Listener close 35
Timing Listener died 140
Timing Listener connection 3678
Timing Listener release 13
Timing Listener run 3258
Timing HTTP_1_0 stream_entity 1723
Timing HTTP_1_0 response 1784
Timing HTTP_1_0 to_store 2347
Timing Listener talk 2501
Timing Listener close 32
Timing Listener release 47
Timing Listener run 3067
Timing Listener died 134
Timing Listener connection 3955
Timing Listener died 35
Timing Listener connection 3820
```

Now the performance has dropped to an average of 261 connections/sec. The time to setup a connection has jumped due to the increased time to setup the time-out. This comes from the extra overhead of the CML operations when there are more events and threads in the server. The time to send a file to

the connection has doubled since this involves lots of message sending which is now slower.

Table 8-5. Timing Measurement Points.

Name	Description
abort request	This measures the time for the server in the Abort module to process a request to add a new time-out.
abort create	This measures the time for the `Abort.create` function to run. It includes the time to send the message to the server without waiting for any reply.
Listener setupConn	This measures the time to create the connection object when a new connection arrives. This mainly consists of setting a socket option and starting the time-out by creating an Abort value.
HTTP_1_0 get	This measures the time to read in the GET request from the socket including parsing all of the headers.
GenNode request	This measures the time for the resource store to forward the request through to the handler. It does not include the time for the directory node handler to run.
HTTP_1_0 stream_entity	This measures the time for the `stream_entity` function to run which transfers the contents of the page to the socket. This includes the time for reading the page from disk.
HTTP_1_0 response	This measures the total time to process a response. This includes the `stream_entity` time above along with the time to write the status and headers.

Name	Description
HTTP_1_0 to_store	This measures the total time to process a request including the time to send it to the store and the time to process the response (above).
Listener talk	This measures the total time to run the HTTP protocol for a connection including all request and response processing.
Listener close	This measures the time to close the socket after the response has been sent.
Listener release	This measures the time to clean-up any temporary files.
Listener run	This measures the total time for a connection including all of the above Listener measurement points.
Listener died	This measures the time to tell the connection manager that a connection has terminated.
Listener connection	This measures the total time for a connection from after the socket's creation until its close.

Notes

1. See RFC1521 for a description of Base64 encoding
2. The URL is `ftp://ftp.hpl.hp.com/pub/httperf`

Chapter 9. The Swerve Detailed Design

Introduction

This chapter discusses the source code in more detail. First I will give an overview of the organisation of the code. Then I will discuss each of the files in detail. I'll describe the files in order from the top (main) down to the low-level code but you'll want to jump around from file to file.

I won't be including all of the source code in this chapter. The discussion will concentrate on what I think are the non-trivial and non-obvious features of the code. You should be reading the source code along with the discussions.

The Organisation of the Code

Perhaps the most awkward thing about the SML module system is that circular dependencies between modules are not allowed. Declarations have to be carefully partitioned between modules to avoid circular dependencies. For example this often means that type declarations have to be factored out to a separate module. This is because you can only have mutually recursive types if they are `datatypes` in the same group of declarations (i.e. joined with the `and` keyword) in the same module. So you end up with a tree of modules to manage.

To help manage the modules I divide them into layers with each layer in its own directory. The module dependencies between layers always point from the top down. Table 9-1 shows the layers of the server and the files in each layer. The layers are shown from the highest to the lowest. The files in each layer are listed in alphabetical order. I usually only place one module in each

file.

Table 9-1. The Module Layers of the Server.

Layer	File (Module)	Purpose
main	main.sml (Main)	This contains the main function for the server.
	startup.sml (Startup)	This contains the code for creating the lock and pid files, setting the uid/gid and reversing this on shutdown.
server	connect.sml (Connect)	This implements a type for a connection along with I/O and utility operations on the connection.
	http10.sml (HTTP_1_0)	This implements the HTTP1.0 protocol over a connection and communicates with the resource store.
	listener.sml (Listener)	This creates the listener socket and accepts new connections.
store	builtin_node.sml (SimpleBuiltinHandler)	This implements some built-in node handlers.
	cgibin_node.sml (CgiNodeHandler)	This implements the CGI node handler.
	dir_node.sml (DirNodeHandler)	This implements nodes that map to files and directories.
	gen_node.sml (GenericNodeFn)	This implements backbone nodes that route requests to handlers. It provides a functor that is specialise by each kind of handler.

Layer	File (Module)	Purpose
	node_auth.sml (NodeAuth)	This implements authorisation checking functions.
	node_factory.sml (NodeFactory)	This implements a function to create resource nodes of different kinds.
	node_handler_sig.sml (NODE_HANDLER)	This defines the signature that a node handler must have. This is the configuration interface between backbone and handler nodes.
	node.sml (Node)	This defines common types for resource nodes. This includes the message protocol between backbone nodes and handlers. Some utility functions are included.
	resp_utils.sml (ResponseUtils)	This implements a few utility functions for creating HTTP responses.
	store.sml (Store)	This is the entry point for the resource store. It builds the node tree and accepts request from the HTTP protocol manager.
ietf	entity.sml (Entity)	This defines a type for a HTTP entity. The entity producer and consumer types and protocol is also included. All of the kinds of producers are implemented in here.

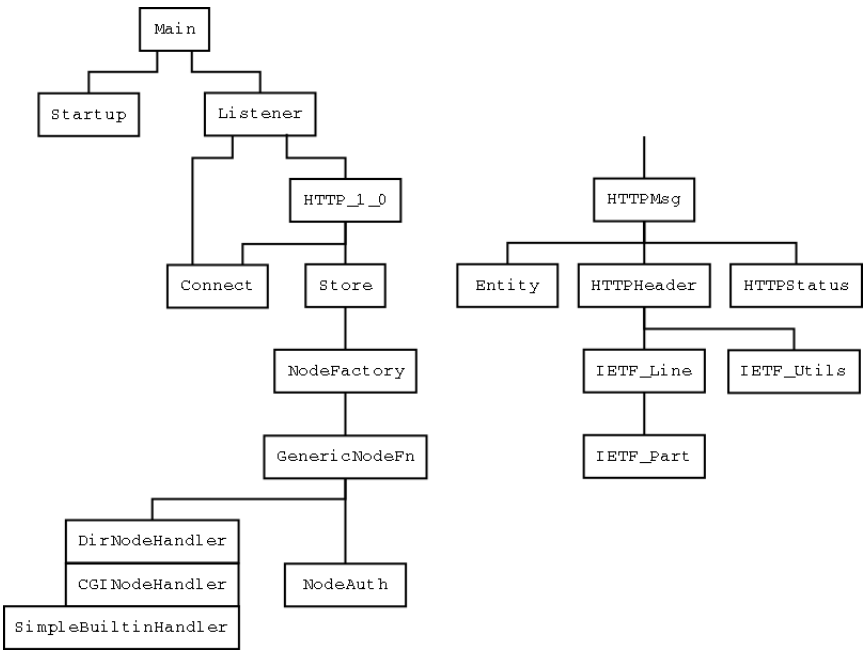
Layer	File (Module)	Purpose
	http_header.sml (HTTPHeader)	This runs the parsing of HTTP headers. A type is defined to represent all of the well-known headers in a fully-parsed format.
	http_msg.sml (HTTPMsg)	This defines the types for a HTTP request and response.
	http_status.sml (HTTPStatus)	This defines the type for a HTTP status code.
	ietf.lex (IETFLex)	This implements a ML-Lex lexer to help parse HTTP headers. It performs all of the regular-expression operations.
	ietf_line.sml (IETF_Line)	This has more functions for parsing HTTP headers. It uses the lexer to split a header line into tokens according the the HTTP specification.
	ietf_part.sml (IETF_Part)	This defines the type for the tokens returned by the functions in ietf_line.sml.
	ietf_utils.sml (IETF_Utils)	This implements a few utility functions. Currently it only has base64 encoding and decoding.
config	config.grm	This is a ML-Yacc grammar for the server's configuration file.
	config.lex	This is a ML-Lex lexer to match the grammar.

Layer	File (Module)	Purpose
	config.sml (Config)	This implements the configuration file parsing. It stores the configuration parameters as global values. Some parameters are pushed down into the common modules to avoid circular dependencies (see `init_globals`).
	config_types.sml (ConfigTypes)	This defines types for the parse tree produced by the grammar.
common	abort.sml (Abort)	This defines a type to represent abort conditions on connections.
	common.sml (Common)	This defines common types and functions that are used all over the place.
	file_io.sml (FileIO)	This implements functions that generally operate on the contents of files, especially wrappers around OS.FileSys functions.
	files.sml (Files)	This implements functions that manage file paths and provide information about files.
	finalise.sml (FinaliseFn)	This implements a functor to provides finalisation of values after garbage collection.
	logging.sml (Log)	This implement the Logging Manager.
	mutex.sml (Mutex)	This implements mutexes for atomic operations on static variables.

Layer	File (Module)	Purpose
	open_mgr.sml	This implements the modules of the Open Manager. This include the open file counter (OpenCounter) and the managers specialised to each kind of openable object such as files and processes.
	profile.sml (MyProfile)	This implements some profiling utilities.
	signals.sml (SignalMgr)	This implements signal catchers. It also catches the pseudo-signal from the garbage collector. The signals are distributed to the rest of the server using multicasted messages.
	singleton.sml (Singleton)	This implements a functor that provides the common pattern of a singleton concurrent object that is started on demand.
	text.sml (TextFrag)	This defines a type for a fragment of text. Fragments can be combined cheaply without copying the text.
	tmpfile.sml (TmpFile)	This implements the management of temporary files including limiting the disk space used and cleaning up when a connection closes.
	url.sml (URL)	This implements a type for representing URLs.

The main dependencies between the modules of the top three levels are shown in Figure 9-1. Only the main interactions are shown, enough to place the modules into a hierarchy. The modules on the right hand side are those in the ietf layer. There would be too many connections joining the store and ietf layers to draw in. I've omitted the config layer as it is fairly simple. The common layer is mostly flat.

Figure 9-1. The Main Dependencies of the Upper Layers.



How to Follow the Code

The following detailed discussions will be organised by layer from the top down. I'm going to describe the modules in a logical order that roughly traces the flow of control for a HTTP request through the server. This will also

mesh with the top-down description of the layers since the functions are calling from the top down through the layers.

This ordering of the discussions means there will be many forward references to modules that will be discussed later, especially those in the the config and common layers. You may want to jump ahead and scan the modules of the common layer first. Especially have a look at the `TextFrag` and `Log` modules in the section called *The Common Layer*.

Throughout the code I've used some common abbreviations for module names.

```
structure Cfg    = Config
structure Hdr    = HTTPHeader
structure Req    = HTTPMsg
structure S      = Socket
structure SS     = Substring
structure Status = HTTPStatus
structure TF     = TextFrag
structure U      = ResponseUtils
```

The `Common` module is always opened for direct access.

Building the Server

Each directory contains a `sources.cm` CM file to control the compilation of the server. Here is the CM file for the `main` directory. (See the section called *Assembling the Hello World Program* in Chapter 2 for more on `CM` files).

```
group is
    /src/smlnj/current/lib/cml.cm
    /src/smlnj/current/lib/cml-lib.cm

    ../common/sources.cm
    ../config/sources.cm
    ../server/sources.cm

    main.sml
    startup.sml
```

This CM file includes all other CM files directly or indirectly so you only need to compile in the `main` directory to compile all of the program.

Note that all uses of `sml-lib` (the SML libary) must be replaced with `cml-lib` in all `.cm` files in all directories. You cannot mix `sml-lib` and `cml-lib` in the same program.

To build the entire program just do the following or use the `Makefile` in the top directory.

```
$ cd main
$ sml
Standard ML of New Jersey, Version 110.0.7, ...
- CM.make();
........
write 5,0: 57 big objects (544 pages) @ 0x1064a0
$
```

The Main Layer

This layer contains the `main` function, command line parsing and the server startup code.

The Main Module

This is a conventional main file with command line processing. Here is the `usage` function to show the command line options.

```
fun usage () = (
    print "Usage: swerve [options]\n";
    print "Options:\n";
    print "  -f file     : (required) specify the server configuration file\n";
    print "  -h          : print this help message\n";
    print "  -T id       : enable test messages according to the id\n";
    print "  -D level    : set the logging level to Debug immediately\n";
    ()
    )
```

The configuration file must be provided and it is immediately parsed. Any errors are written to `stdout` during the parsing.

The `main` function starts up CML and jumps to the command line processing. I have to set up the multicast channels in the signal manager before the configuration file is parsed. Otherwise the file I/O handling with finalisation, which depends on the garbage collector signal, will hang when attaching to the signal.

```
fun main(arg0, argv) =
let
    fun run() =
    (
        SignalMgr.init();   (* required by OpenMgr *)
        process_args argv
    )
in
    RunCML.doit(run, NONE)
end

val _ = SMLofNJ.exportFn("swerve", main)
```

After the command line options have been processed comes the `run` function.

```
fun run() =
let
in
    TraceCML.setUncaughtFn fatal_exception;
    (* TraceCML.setTraceFile TraceCML.TraceToErr; *)

    StartUp.startup();

    (*  Make CML run our finish() function at every shutdown. *)
    ignore(RunCML.addCleaner("finish", [RunCML.AtShutdown],
                    fn _ => StartUp.finish()));

    Listener.run();
    success()               (* shouldn't get here *)
end
```

The two main-line steps are to run the startup code and then start the listener. I also arrange for the `StartUp.finish` function to be run when CML is shutdown. CML has a system of "cleaner" functions that can be run

at various points. The `AtShutdown` point ensures that the cleaner will be run whether the server exits normally or fails on a fatal exception. The only way the cleaner won't run is a crash of the run-time. More information on cleaners can be found in the CML source code in the file `core-cml/cml-cleanup-sig.sml`.

The `run` function never returns normally. Instead the server is shutdown by calling the `RunCML.shutdown` function. This is done through either of the `success` or `fail` functions in the `Common` module. If there is an exception out of the `run` function then it will be caught in the `process_args` function which will call `fail`.

The `run` function also contains some commented-out debugging code which uses the `TraceCML` module. I used this during debugging to trace the termination of threads. For an example see the `HTTP_1_0` module. Uncaught exceptions in a thread are reported via the `fatal_exception` function (not shown here).

The Startup Module

Here is the `startup` function.

```
fun startup() =
let
in
    MyProfile.start();

    if Cfg.haveServerConfig()
    then
        ()
    else
    (
        Log.error ["The server configuration has not been specified."];
        raise FatalX
    );

    (*  Give up if there have been errors already. *)
    if Log.numErrors() > 0 then raise FatalX else ();
```

```
    (*  The configuration code checks that all of the files and
        directories exist.
    *)
    setuid();
    create_lock();

    (*  Give up again. *)
    if Log.numErrors() > 0 then raise FatalX else ();

    ()
end
```

It checks that the configuration has been successfully read. If the configuration file was not specified or if there were any errors while processing the configuration then the server will exit with a fatal error. The `FatalX` exception is defined in the `Common` module and caught as explained above.

Next the `startup` function sets the user and group ids if configured and creates the lock and pid files. If there are any more errors from doing this then it is also a fatal error. The `Startup.finish` function, called at shutdown time, removes the lock and pid files. I'll skip the code for setting the ids and show the locking functions.

```
and create_lock() =
let
    val Cfg.ServerConfig {var_dir, ...} = Cfg.getServerConfig()
    val lock_file = Files.appendFile var_dir "lock"
    val pid_file  = Files.appendFile var_dir "pid"
in
    Log.inform Log.Debug (fn() => TF.L ["Creating lock file ", lock_file]);

    if FileIO.exclCreate lock_file
    then
        let
            val strm = TextIO.openOut pid_file
            val pid = Posix.ProcEnv.getpid()
            val w   = Posix.Process.pidToWord pid
        in
            TextIO.output(strm, SysWord.fmt StringCvt.DEC w);
            TextIO.output(strm, "\n");
            TextIO.closeOut(strm)
        end
```

```
        handle x => (Log.logExn x; raise x)
    else
    (
        Log.error ["Another server is already running."];
        raise FatalX
    )
end
handle _ => raise FatalX



and remove_lock() =
let
    val Cfg.ServerConfig {var_dir, ...} = Cfg.getServerConfig()
    val lock_file = Files.appendFile var_dir "lock"
    val pid_file  = Files.appendFile var_dir "pid"
in
    FileIO.removeFile pid_file;
    FileIO.removeFile lock_file
end
```

Here is the `FileIO.exclCreate` function.

```
fun exclCreate file =
(
    IO.close(FS.createf(file, FS.O_WRONLY, FS.O.excl,
                FS.S.flags[FS.S.irusr, FS.S.iwusr]));
    true
)
handle
  x as OS.SysErr (_, eopt) =>
    (if isSome eopt andalso valOf eopt = Posix.Error.exist
     then
        false       (* failed to exclusively create *)
     else
        (Log.logExnArg file x; raise x) (* failed with error *)
    )

| x => (Log.logExnArg file x; raise x)
```

I've settled for creating a lock file with the `Posix.FileSys.createf` function using the `excl` flag. In UNIX terms this means using the `open` system call with the `O_CREAT`, `O_WRONLY` and `O_EXCL` flags and mode 0600. This will work fine as long as the directory containing the lock file is not

mounted via NFS. I've made it a requirement in the server configuration that this be so.

I can check for the `EEXIST` errno code by catching the `OS.SysErr` exception. It just so happens that the `OS.syserror` type it contains is the same as the `Posix.Error.syserror` type and the `Posix.Error` module contains example values for each error code.

Any other error while creating the lock file will be logged and propagated as an exception.

Once the lock file is created I can write the process' pid into a file straightforwardly. The only tricky bit is tracking down the right type conversion functions. The SML basis documentation doesn't make it explicit that the `Posix.ProcEnv.pid` type is the same as the `Posix.Process.pid` type.

The Server Layer

This layer contains code for managing the sockets and running the HTTP protocol over a client connection.

The Listener Module

As explained in the section called *The Architecture of the Server* in Chapter 8 each socket is handled by a separate thread. The listener thread waits for a new connection and then spawns a thread for the new connection.

The listener thread counts the connection threads and refuses new connections when the configured limit is reached. There are two ways to implement this. In both cases the central issue is how the listener thread discovers when a connection has terminated. The straightforward approach

is to wait on a join event for each connection thread together with an accept event for a new connection. The code would be something like this.

```
fun server threads =
let
    fun join thread =
    let
        fun keep t = not(CML.sameTid(t, thread))
        fun remove() = List.filter keep threads
    in
        CML.wrap(CML.joinEvt thread, remove)
    end

    val ac_evt = CML.wrap(Socket.acceptEvt listener, new_connection threads)
    val join_evts = map join threads

    val new_threads = CML.select (ac_evt::join_evts)
in
    server new_threads
end
```

This is a server loop for a listener. The state is the list of connection threads. Here the `join` function associates each connection thread with a `remove` function that removes the thread from the list. At the same time it waits for an `accept` event on the listening socket. (The `acceptEvt` function is a CML extension of the Socket module).

My concern with this implementation is the overhead if the number of connections is large. If I dream of my server one day running a site with hundreds of connections then I'm not keen on all this processing over long lists of threads. All I really need to do is count the connection threads. So I have a connection thread send a message to the listener thread when it terminates. Here is the body of the listener thread which runs as the main thread of the server.

```
and serve listener max_clients conn_timeout =
let
    val lchan: ListenMsg CML.chan = CML.channel()

    fun loop num_connects =
    let
        (*  If we have too many then we will refuse the new
```

```
            connection.  We require each connection thread to tell
            us when it dies.

            We won't log the connection refusals to avoid the log
            overflowing on a DOS attack.
        *)
        fun new_connect (conn, conn_addr) =
        (
            if (isSome max_clients) andalso
                    num_connects >= (valOf max_clients)
            then
            (
                Socket.close conn;
                num_connects
            )
            else
            (
                FileIO.setCloseExec(Socket.pollDesc conn);

                CML.spawn(MyProfile.timeIt "Listener connection"
                    (connection lchan conn conn_addr conn_timeout));

                num_connects+1
            )
        )
        handle x =>
            (
                (Socket.close conn) handle _ => ();
                Log.logExn x;
                num_connects
            )

        fun msg ConnDied = num_connects - 1

        val new_num = CML.select[
            CML.wrap(S.acceptEvt listener, new_connect),
            CML.wrap(CML.recvEvt lchan, msg)
            ]
    in
        loop new_num
    end
in
    loop 0
end
handle x =>
    (
        Socket.close listener;
```

```
        Log.logExn x;
        raise FatalX
    )
```

The arguments `max_clients` and `conn_timeout` are configuration parameters and `listener` is the socket to listen on. The configuration parameters are integer option values from the `Config.ServerConfig` type. (See the `run` function below).

The `loop` function implements a state machine with the number of connections as the state. The action happens at the call to `CML.select`. This waits for either a new connection on the listener socket or a message over the manager channel. The result from the event dispatch is a new state value, the new number of connections.

When a new connection arrives it is accepted and the socket and client's IP address are passed to the `new_connect` function. If a maximum number of clients is configured and the limit is exceeded then the connection is immediately closed. There is no change to the number of connections in this case. I could send back a HTTP status saying "503 Service Unavailable" but it is legal to just close the connection. If I have a connection limit then I'm worried about the load on the server and I won't want to waste more time telling the client to go away nicely.

If the connection is accepted then I spawn a thread to run the connection. The socket is marked as close-on-exec so that CGI child processes don't inherit it.

If there is an exception at this stage then I log it and close the channel. I'm careful here to not let another exception break the server loop.

If a `ConnDied` message comes in from a connection thread then this just decrements the number of connections.

Here is the `run` function which is called from main. It creates the listener socket and calls the above `serve` function.

```
fun run() =
let
```

```
    val ServerConfig {
            conn_timeout: int option,
            max_clients:  int option,
            listen_host,
            listen_port,
            ...
            } = getServerConfig()

    (*  Build an address for listening.
    *)
    val listen_addr =
        case listen_host of
          NONE => INetSock.any listen_port

        | SOME host =>
        (
            (*  The configuration is supposed to have validated
                the host.
            *)
            case NetHostDB.getByName host of
              NONE       => raise InternalError "invalid host"
            | SOME entry =>
                INetSock.toAddr(NetHostDB.addr entry, listen_port)
        )

    val listener = INetSock.TCP.socket()
in
    (*  Doing these fixes the type of the socket as passive. *)
    Socket.Ctl.setREUSEADDR(listener, true);
    Socket.bind(listener, listen_addr);
    Socket.listen(listener, 9);
    FileIO.setCloseExec(Socket.pollDesc listener);

    serve listener max_clients conn_timeout
end
handle x => (Log.logExn x; raise FatalX)
```

The server's configuration provides a port to listen on and optionally an IP address to bind the port to. This could be "localhost" for private use but it could be the address of a particular interface on a server. The `getByName` function is the equivalent of the C `gethostbyname()` function. The `INetSock.toAddr` function constructs the address for an internet socket. This address value is equivalent to the C `sockaddr_in` struct. The remainder of the steps are conventional for setting up a listener socket.

It's worth looking again at how the type of the listener socket is fixed by these function calls (see the section called *The Specific Socket Types* in Chapter 4). The `INetSock.TCP.socket` function involves these types in the INetSock structure. (I've added parentheses in `stream_sock` to show the precedence).

```
...
type 'a sock = (inet, 'a) Socket.sock
type 'a stream_sock = ('a Socket.stream) sock
...
structure TCP : sig
    val socket  : unit -> 'a stream_sock
    ...
end
```

Substituting these type equations gives the intermediate type for the socket as

```
(INetSock.inet, 'a Socket.stream) Socket.sock
```

In this type `'a` is a place holder for the passive/active mode. The type of the `listen` function will constrain this variable to be `Socket.passive` giving the final type for a listening socket.

```
(INetSock.inet, Socket.passive Socket.stream) Socket.sock
```

This is the only type that the `Socket.accept` function will accept. So the type system ensures that you don't forget to call the `listen` function and similarly you can't accidentally accept on a connected socket.

Finally here is the code that controls the connection.

```
and connection lchan sock sock_addr conn_timeout () =
let
    fun run() =
    let
        val conn = MyProfile.timeIt "Listener setupConn"
                    Connect.setupConn{
                        socket  = sock,
                        sock_addr = sock_addr,
                        timeout = conn_timeout
                        }
```

```
    in
        MyProfile.timeIt "Listener talk"
            HTTP_1_0.talk conn;

        MyProfile.timeIt "Listener close"
            Connect.close conn;

        Log.testInform G.TestConnect Log.Debug
            (fn()=>TF.S "Connection closed");

        MyProfile.timeIt "Listener release"
            TmpFile.releasePort(Connect.getPort conn);

        Log.testInform G.TestConnect Log.Debug
            (fn()=>TF.S "TmpFiles released")
    end

in
    Log.inform Log.Info (fn()=>TF.C [TF.S "New connection from ",
                                format_addr sock_addr]);

    MyProfile.timeIt "Listener run"
        run();

    Log.inform Log.Info (fn()=>TF.C [TF.S "End of connection from ",
                                format_addr sock_addr]);

    MyProfile.timeIt "Listener died"
        CML.send(lchan, ConnDied)
end
handle x =>
    let
        (*  See also Connect.getPort *)
        val (_, port) = INetSock.fromAddr sock_addr
    in
        (
            Socket.close sock;
            TmpFile.releasePort port
        ) handle _ => ();           (* being paranoid *)
        Log.logExn x;
        CML.send(lchan, ConnDied)
    end
```

Most of the bulk of the code is for handling contingencies. (The `timeIt` calls are there for performance testing). Essentially it runs the `talk` function in

the HTTP protocol module and then closes the connection. The `Connect.setupConn` function wraps up the socket details into a connection value. It also starts a time-out for the connection if the server is configured for this. This time-out applies to the entire interval from the acceptance of the connection through to the closing including the running of any CGI scripts. An overview of the time-out handling can be found in the section called *Time-outs* in Chapter 8.

The `TmpFile.releasePort` function deletes any temporary files that have been created for the connection. They would typically contain posted HTTP entities for CGI scripts to read. The cleanup must be carefully repeated if any exception is caught. No exceptions can be allowed to get around the connection thread sending the `ConnDied` message or else the listener thread would slowly leak connection capacity.

The Connect Module

This module contains functions for I/O over a connection socket while looking out for a time-out condition.

The header of a HTTP request is line-oriented but there is no defined line-length limit. Even if there was, a robust server must be able to cope with arbitrarily long lines without a "buffer overflow" or filling memory. So I've decided on a line-length limit of 10000 which should be enough for header lines. Characters beyond the limit are discarded.

Since lines are usually terminated with a CR-LF sequence, custom line-reading code is required. This requires a buffer to accumulate chunks of characters from the socket until a complete line is received. See the `readLine` function for the code for the line splitting. In the rest of this section I'll only describe the lower-level details.

Here is the type for a connection.

```
datatype Conn = Conn of {
        socket:     Socket.active INetSock.stream_sock,
```

```
        port:       int,
        addr:       NetHostDB.in_addr,

        is_open:    bool ref,
        rdbuf:      string ref,
        rdlen:      int ref,        (* number of chars left *)
        rdoff:      int ref,        (* offset to next avail char *)

        (*  This transmits abort messages to all receivers. *)
        abort:      Abort.Abort
        }
```

The `rdbuf` field is a string buffer that is updated in place. The `rdlen` and `rdoff` fields point to a range of characters in the buffer that have not be processed yet. The `abort` field propagates a time-out condition to any party interested in the connection. Here is the function to create a connection value.

```
fun setupConn {socket, sock_addr, timeout} =
let
    (*  Apache has special linger handling but SO_LINGER works
        on Linux.
    *)
    val _ = S.Ctl.setLINGER(socket, SOME(Time.fromSeconds 2))

    val (addr, port) = INetSock.fromAddr sock_addr

    val abort =
        case timeout of
          NONE   => Abort.never()
        | SOME t => Abort.create t
in
    Conn {
        socket  = socket,
        port    = port,
        addr    = addr,
        is_open = ref true,
        rdbuf   = ref "",
        rdlen   = ref 0,
        rdoff   = ref 0,
        abort   = abort
        }
end
```

The LINGER option makes a close of the socket wait until the socket has finished sending all of the response back to the client (or until the 2 second time-out I've specified is reached). The alternative is that the close returns immediately and the socket drains in the background. But in this case during this draining it would not be counted against the server's connection limit. You could imagine a busy server accumulating an unlimited number of lingering sockets if they weren't counted.

The `Abort.create` function creates an abort object for the given time-out value. If no time-out is required then `Abort.never` creates a similar object that never times-out but can still be forced into the time-out state. If the connection is broken a time-out is forced so the server only has to test for the one condition.

The I/O functions raise the `Timeout` exception if they detect an attempt to read or write after a time-out. Here is the `fill_buf` function which is the core of the reading code. The various reading functions call `fill_buf` to get the next chunk of characters from the socket.

```
and fill_buf (Conn {socket, rdbuf, rdlen, rdoff, abort, ...}) =
let
    fun takeVec v =
    let
        val s = Byte.bytesToString v
    in
        rdbuf := s;
        rdlen := size s;
        rdoff := 0
    end
in
    CML.select [
        CML.wrap(Abort.evt abort, (fn() => raise Timeout)),
        CML.wrap(S.recvVecEvt(socket, 1024), takeVec)
        ]
end
```

It waits for a chunk of up to 1024 characters from the socket or until a time-out. The chunk is a vector of bytes which I convert to a string and place into the buffer. The `Byte.bytesToString` and `Byte.stringToBytes`

functions are actually internally just type casts between a vector of bytes and a vector of characters. They don't have any run-time cost.

The opposite is the `write` function to send a string to a socket.

```
and write (conn as Conn {socket, is_open, ...}) msg =
(
    if !is_open
    then
    let
        val bytes = Byte.stringToBytes msg
        val len   = size msg

        (*  n is the number of bytes written so far. *)
        fun loop n =
        (
            if aborted conn
            then
                raise Timeout
            else
            let
                (* val _ = toErr(concat["Connect.write sendVec n=",
                            Int.toString n, " len=",
                            Int.toString len, "\n"]) *)
                val buf = {buf=bytes, i=n, sz=NONE}
                val sent = n + (S.sendVec(socket, buf))
            in
                if sent >= len
                then
                    ()
                else
                    loop sent
            end
        )
    in
        loop 0
    end
    else
        ()
)
```

Remember from the section called *Time-outs* in Chapter 8 that after a time-out some of the request processing may linger until the garbage collector cleans it up. The time-out will quickly force the socket to be closed via the `Timeout` exception being propagated into the HTTP_1_0 module.

But the connection object may linger for some time and there may be further attempts to write to the connection. So all I/O functions check that the socket is still open and there hasn't been an abort condition before proceeding.

When sending to the socket there is the risk of a partial write. I need a loop to keep sending until all of the string is sent. The `sendVec` function makes it easy to send a message in chunks using the `buf` record type. A time-out is checked before each attempt.

The HTTP_1_0 Module

This module runs the HTTP version 1.0 protocol. This consists of reading in and parsing a request from the connection socket and writing back the response. It exports the one function `talk`.

```
fun talk conn =
let
    val req = MyProfile.timeIt "HTTP_1_0 get" get_request conn
in
    if G.testing G.TestShowRequest
    then
        (Req.dumpRequest req)
    else
        ();

    MyProfile.timeIt "HTTP_1_0 to_store" (fn()=>to_store conn req) ()
end
handle Bad status => send_status conn status
```

This just gets the request and sends it to the resource store. The store is expected to send a response back at some later time. If there is an error while reading the request the `Bad` exception will be raised and it will contain a status that can be sent back to the client. Usually this is just the "400 Bad Request" or "500 Server Fail" status.

Here is the `get_request` function. It reads the parts of a request in a straight-forward manner and builds the `Request` value.

```
and get_request conn : Req.Request =
```

```
let
    val (method, url, protocol) = get_request_line conn
    val headers = get_all_headers conn
    val entity  = get_entity headers conn
in
    Log.testInform G.TestShowRequest Log.Debug
        (fn()=>TF.S "got a request");

    Req.Request {
        method  = method,
        url     = url,
        protocol= protocol,
        headers = headers,
        entity  = entity,

        port    = Connect.getPort conn,
        client  = Connect.getAddress conn,

        rvar    = Sy.iVar(),
        abort   = Connect.getAbort conn
        }
end
```

I won't show the `get_request_line` function as it is just a simple bit of string splitting. The `get_all_headers` function is just a wrapper for the `readAllHeaders` function of the `HTTP_Header` module (see the section called *The HTTPHeader Module*). The `get_entity` function is more interesting.

```
and get_entity headers conn : Entity.Entity =
let
    val Config.ServerConfig {max_req_size, ...} = Config.getServerConfig()
    val chunk_size = 8192

    fun read_file len =
    let
        val _ = Log.testInform G.TestShowRequest Log.Debug
            (fn()=>TF.L ["HTTP reading into file len=",
                         Int.toString len])

        val (tmp_file, writer) = create_body_file conn len
        val strm = BinIOWriter.get writer

        fun loop 0 = ()
        |   loop n =
```

```
        (
            case Connect.read conn chunk_size of
              NONE        => Log.log Log.Warn (TF.S "short body")
            | SOME (s, _) =>
            (
                BinIO.output(strm, Byte.stringToBytes s);
                loop (n-(size s))
            )
        )
    in
        loop len;
        BinIOWriter.closeIt writer;
        Entity.tmpProducer tmp_file
    end
    handle x => (Log.logExn x; raise Bad Status.ServerFail)


    fun read_mem len =
    let
        val _ = Log.testInform G.TestShowRequest Log.Debug
            (fn()=>TF.L ["HTTP reading into mem len=",
                         Int.toString len])

        val (frag, _) = Connect.readAll conn len
    in
        Entity.textProducer frag
    end
    handle x => (Log.logExn x; raise Bad Status.ServerFail)


    (*  ReqTooLarge is v1.1 only but it's too good to avoid. *)
    fun check_req_limit len =
    (
        case max_req_size of
          NONE   => ()
        | SOME m => if len > m then
                        raise Bad Status.ReqTooLarge else ()
    )

    val einfo = Hdr.toEntityInfo headers
    val Entity.Info {length, ...} = einfo
in
    case length of
      NONE   => Entity.None

    | SOME n =>
        let
```

```
            val () = check_req_limit n

            val body =
                if n > body_limit
                then
                    read_file n
                else
                    read_mem n
        in
            Entity.Entity {
                info    = einfo,
                body    = body
                }
        end
end
```

All of the entity body is read in. First the headers are studied to get those relevant to the entity, in particular its length. If the length is 10000 (`body_limit`) bytes or less then I copy it into a string in memory. Everything from the socket is read in and passed to the `textProducer` function. This creates an entity with a producer function (see the section called *Entities, Producers and Consumers* in Chapter 8) which can deliver the content of the string.

If the file is larger than 10000 bytes then I copy it to a temporary file. Temporary files go into the directory specified by the `TmpDir` configuration parameter (see the section called *The Server Parameters* in Chapter 8). The file name includes the port number so that it is easy to clean up all temporary files associated with a connection. The `create_body_file` function (below) will create and open the temporary file. A loop transfers the entity to the file in chunks. It reads only exactly the number of bytes expected from the Length header. An entity is created with a producer that can deliver from a temporary file.

Here is the `create_body_file` function. It will block until there is enough disk space and file descriptors for the write to proceed. See the section called *The Open File Manager* for more details on this. The blocking may be aborted by a time-out condition.

```
and create_body_file conn len :(TmpFile.TmpFile * BinIOWriter.Holder)=
```

```
let
    val Config.ServerConfig {tmp_dir, ...} = Config.getServerConfig()
    val port  = Connect.getPort conn
    val abort = Connect.getAbort conn
in
    case TmpFile.newBodyFile abort tmp_dir len port of
      (* errors have already been logged *)
      NONE => raise Bad Status.ServerFail

    | SOME tmp =>
        (tmp, valOf(BinIOWriter.openIt abort (TmpFile.getName tmp)))
            handle x => raise Bad Status.ServerFail
end
```

Once the request is read it is send off to the resource store.

```
and to_store conn req =
let
    val Req.Request {rvar, abort, ...} = req
in
    Log.testInform G.TestStoreProto Log.Debug
        (fn()=>TF.S "HTTP: sending to the store");

    Store.deliver req;

    (*  Get a response or do nothing if there is an abort condition.
    *)
    CML.select[
        CML.wrap(Abort.evt abort, fn () => ()),

        CML.wrap(Sy.iGetEvt rvar,
                MyProfile.timeIt "HTTP_1_0 response"
                    (handle_response conn req))
        ]
end
```

The `to_store` function delivers it to the store and then blocks waiting for a response. The store processes multiple requests concurrently so it must send the response over the reply channel when it is ready. If a response is received it goes to `handle_response`. If there is a time-out before the response comes back then nothing is done and `to_store` returns immediately to the `talk` function which returns to the connection handler with nothing written

to the socket. (Note that I haven't implemented Redirect requests from CGI scripts yet.)

A normal response is handled by this function.

```
and handle_response conn req response : unit =
let
    val Req.Request {method, abort, ...} = req
    val Req.Response {status, headers, entity} = response
in
    Log.testInform G.TestShowResponse Log.Debug
        (fn()=>TF.S "HTTP Protocol got a response");

    (
        send_status  conn status;
        send_headers conn headers;

        MyProfile.timeIt "HTTP_1_0 stream_entity"
            (fn() => stream_entity abort conn entity
                    (method = Req.HEAD)
                    (Status.needsBody status))
            ()
    )
    handle
      Connect.Timeout => (Abort.force abort)
    | x => (Log.logExn x; Abort.force abort)
end
```

It just delivers the parts of the response to the connection: the status, headers and entity. Since there is writing to the connection there may be a `Timeout` exception (see the section called *The Connect Module*). I force the abort condition on any exception to make sure that it is broadcast to all interested parties.

Sending the status and headers of the response is straight-forward. Sending the entity is more interesting. As explained in the section called *The Connection Protocol* in Chapter 8 the entity is streamed out using a pair of producer and consumer. Here is the `stream_entity` function.

```
and stream_entity abort conn entity head_method needs_body =
let
    val csmr: Entity.Consumer = CML.channel()
```

```
    fun receiver() =
    (
        case CML.recv csmr of
          Entity.XferInfo info   => (send_info info; receiver())
        | Entity.XferBytes bytes => (send_bytes bytes; receiver())
        | Entity.XferDone        => ()
        | Entity.XferAbort       => ()
    )


    (*  Send the entity headers. *)
    and send_info info =
    let
        val hdrs = from_entity_info info
    in
        send_headers conn hdrs;
        end_headers conn
    end


    and send_bytes bytes =
    (
        if head_method
        then
            ()
        else
            Connect.write conn (Byte.bytesToString  bytes)
    )
in
    case entity of
      Entity.None =>
        (
            if needs_body                   (* see RFC1945 7.2 *)
            then
                Connect.write conn "Content-Length: 0\r\n"
            else
                ();
            end_headers conn
        )

    | _ =>
    let
        val pthread = Entity.startProducer abort entity csmr
        (* val _  = TraceCML.watch("producer", pthread) *)
    in
        (*  Don't skip the join. The producer must be allowed to
            clean up a CGI process nicely.
```

```
        *)
        receiver() handle x => Log.logExn x;
        CML.sync(CML.joinEvt pthread)  (* wait for producer to stop *)
    end
end
```

The `receiver` function behaves as a consumer receiving the entity transfer protocol over the `csmr` channel. Down the bottom of the function is the call to `Entity.startProducer`, for non-empty entities. This spawns a new thread to run the producer function. The receiver runs in the thread of the HTTP protocol code which is the same thread as manages the connection.

The `head_method` flag indicates that the request used the HEAD method therefore the body of the entity must be suppressed. It is necessary to run the transfer protocol to the end so that the producer, which may be a CGI script, can terminate properly. The `needs_body` flag indicates that the status code that is being returned requires an entity body. This is true for all informational statuses (in the 1xx range) and also "204 No Content" and "304 Not Modified". So if the entity happens not to have a body I have to insert an empty one.

The Store Layer

This layer implements the resource store with all of the handlers. I describe the modules in roughly logical order.

The Store Module

This is the front door to the resource store. It exports one function for the HTTP protocol module to deliver a request.

```
and deliver req =
let
    val Req.Request  {url, ...}  = req
```

```
    val URL.HTTP_URL {path, ...} = url

    val msg = Node.HTTPRequest {
                request = req,
                segs    = Cfg.URLPathToList path
                }
in
    Node.send (get_root()) msg
end
```

The request is packaged along with an abstraction of the list of segments in the path of the URL. This segment list is used for routing the message through the store. Then the message is sent on to the root node.

A node is implemented by a backbone thread as described in the section called *Resource Store Nodes* in Chapter 8. Here is the type definition.

```
datatype Node = Node of {
        name:       string,
        in_mbox:    NodeMsg Mailbox.mbox
        }
```

The input to the node is a mailbox which has unlimited buffering. This prevents a slow resource node from congesting the tree above it. I am relying on the server's connection limit to prevent these mailboxes from filling with a huge number of messages if a resource becomes congested. Each node also has a name which is the same as the last segment of the path that leads to the node. Message routing is done by comparing the segments in the message with the names of the nodes.

The root node is stored as a singleton object. The resource tree is built on demand by the `build_node_tree` function when the first request arrives.

```
and build_node_tree() : Node.Node =
let
    fun add_paths (c as Cfg.NodeConfig {path, ...}) = (path, c)

    val all_configs = Cfg.getNodeConfigs()
    val with_paths  = map add_paths all_configs
    val root        = build_level [] (Node.initOptions false) with_paths
in
    root
```

```
end
```

The `build_node_tree` function is just a wrapper around the `build_level` function which recurses to build all of the levels of the tree.

```
and build_level path options config_pairs : Node.Node =
let
    val () = Log.testInform Globals.TestStoreBuild Log.Debug
                (fn() => TF.L ["Installing resource ",
                               Cfg.listToString path]);

    type Pair = (string list * Cfg.NodeConfig)

    val table: Pair list STRT.hash_table = STRT.mkTable(101, NotFound)

    fun add (remainder, config) =
    (
        case remainder of
          [] => ()

        | (h::t) =>
        (
            case STRT.find table h of
              NONE     => STRT.insert table (h, [(t, config)])
            | SOME lst => STRT.insert table (h, (t, config)::lst)
        )
    )

    (*  If there is no config for this node then we fake a node
        that always rejects an attempt to access it.

        There may not be a config if this level is an intermediate segment
        of a config path.  E.g. node /cgi/env {...}
    *)
    val self_config =
        case Cfg.findNodeConfig path of
          NONE   => U.builtinConfig path "reject"
        | SOME c => c


    (*  Compute the option flags for this node.
    *)
    val self_options =
        let
            val Cfg.NodeConfig {options = formula, ...} = self_config
        in
```

```
            Node.performFormula options formula
        end


    fun build (name, pairs) =
    let
        val prefix = path @ [name]
    in
        build_level prefix self_options pairs
    end

    val () = app add config_pairs;
    val items: (string * Pair list) list = STRT.listItemsi table
    val children = map build items

    val the_node =
            NodeFactory.create {
                config    = self_config,
                children  = children,
                options   = self_options
                }
in
    getOpt(the_node, make_rejecter path)
end
```

The `path` argument is the path down to the node being built. The `options` argument is a set of node options derived from the options in the server configuration file. The inheritance of options between nodes is performed during this building process.

The `config_pairs` argument is a list of node configurations for all nodes that will be in the sub-tree below the node being built. The path from each configuration has been separated out into a pair with the configuration. The path is in the form of a list of strings and is relative to the node being built. So for example for a node with path `/a/b/c` at the initial call to `build_level`, the pair will be `(["a", "b", "c"], config)`.

The algorithm is to sort the node configurations according to the first part of their paths and group them according to this part. For example for the paths `/a/b/c`, `/a/d` and `/e` I want these groups:

```
"a"     ->      [ (["b", "c"], config1), (["d"], config2) ]
"e"     ->      [ ([], config3) ]
```

This tells me that there will be two child nodes named "a" and "e". The "a" node will in turn have children with sub-paths of `b/c` and `d`. The "e" node will have no children. I've implemented the grouping using a hash table. It maps from the leading part to a list of those pairs that share the path. The `add` function inserts each configuration pair into the table. The `listItemsi` function extracts all of the entries from the table in the form of pairs of key and value where the key is the leading part and the value is the list of pairs. These are the groups. For each group I build a child node by recursing into `build_level`.

Once I have the child nodes I can build the node in question by calling the node factory (see the section called *The Node Factory*). The configuration for the node, in `self_config`, is looked up from the path being built. This checks that the path corresponds to a real node. For example if the server's configuration only contains the paths `/a` and `/a/b/c` then there is no node that corresponds to the path `/a/b` and it won't have a configuration. In this case a dummy node is built for the path `/a/b` that just rejects all requests to it. The `builtinConfig` function creates a dummy configuration for the rejecting node.

```
and make_rejecter path =
let
    val node =
        NodeFactory.create {
            config    = U.builtinConfig path "reject",
            children  = [],
            options   = Node.initOptions false
            }
in
    case node of
      NONE => (     (* something wrong if we can't do this *)
        Log.fatal ["Cannot create a rejecter node."];
        Common.fail()
        )

    | SOME n => n
end
```

If the factory fails to create the node and returns NONE then I try again to make a node with the dummy rejecting configuration. If this fails then the server is not able to make any kind of node and it gives up with a fatal error.

The Node Factory

The NodeFactory module encapsulates the creating of different kinds of resource nodes. The factory function is described by the following type from the Node module.

```
type NodeCreator = {
        config:     Config.NodeConfig,  (* URL path that reaches this node. *)
        children:   Node list,          (* child nodes *)
        options:    Options
        } ->
        Node option
```

It creates a node given its configuration and its children (so the tree of nodes is built from the bottom up). The options are flags that are derived from the node configuration (see the section called *The Node Parameters* in Chapter 8). The caller is responsible for performing the inheritance of options from parent nodes, for example see the store building in the section called *The Store Module*).

```
datatype Options = Options of {
        exec_cgi:       bool,
        follow_sym:     bool,
        with_subdirs:   bool
        }
```

It is in the NodeFactory module that the different kinds of nodes are assembled.

```
structure DirNode = GenericNodeFn(
                structure Handler = DirNodeHandler)

structure CgiNode = GenericNodeFn(
                structure Handler = CgiNodeHandler)
```

```
structure SimpleBuiltinNode = GenericNodeFn(
                structure Handler = SimpleBuiltinHandler)
```

Each kind is assembled from the generic node functor, which implements the backbone thread, and the handler which implements the handler thread. This allows the backbone thread to be specialised to the needs of the handler (see the section called *The Generic Node*).

Here is the `create` function of the factory.

```
fun create {config, children, options} =
let
    val Cfg.NodeConfig {path, kind, ...} = config

    val () = Log.testInform Globals.TestStoreBuild Log.Debug
            (fn() => TF.L ["Creating node ", Cfg.listToString path]);

    (*  This name is used for locating children from path segments.
        See GenericNodeFn.forward_child.
    *)
    val node_name = if null path then "/" else List.last path
in
    case kind of
      Cfg.NodeIsDir {path} =>
        DirNode.create {
            name     = node_name,
            arg      = path,
            config   = config,
            options  = options,
            factory  = create,
            children = children
            }

    | Cfg.NodeIsBuiltin {name} =>
        SimpleBuiltinNode.create {
            name     = node_name,
            arg      = name,
            config   = config,
            options  = options,
            factory  = create,
            children = children
            }

    | Cfg.NodeIsScript {path} =>
        CgiNode.create {
```

```
            name     = node_name,
            arg      = path,
            config   = config,
            options  = options,
            factory  = create,
            children = children
            }
end
```

What is note-worthy here is that the factory is itself passed down to the create functions for each kind of node. This gets around a problem of circular dependencies between modules. The directory node handler needs to be able to create nodes on demand for the sub-directories that it encounters. So it wants to call the factory. But the factory must be able to call the `create` function for the directory node. My solution to this is to pass the factory's `create` function down to the directory node handler so that it can call it back (call up).

The Generic Node

The GenericNodeFn generic node functor exports a `create` function to create a store node. Each kind of node takes its own type of extra arguments when creating a node. For example the CGI node needs the path to the CGI script. This means that the type of the create function varies with the kind of handler. This tells me to use a functor and specialise it with the handler.

```
functor GenericNodeFn(
    structure Handler: NODE_HANDLER
    ): GENERIC_NODE =
struct
```

Here is the signature that a handler module must export. The comments say it all.

```
signature NODE_HANDLER =
sig

    (*  A value of this type is passed to the create function for the
```

```
        handler.
    *)
    type CreateArg

    (*  Create the thread for the handler. Optionally a new
        node configuration can be returned to update the original
        configuration.  All security-related initialisation must be
        done in here so that the master node will be blocked until
        it is ready.

        The caller should be prepared to handle exceptions from here if
        the creation fails.
    *)
    val init:   CreateArg -> Node.HndMbox * (Config.SwerveConfig option)


    (*  This tests if the handler will take the last segment
        of the URL path. For example a directory node wants
        the last segment as a file name.
    *)
    val canTakeLast:    Config.NodeConfig -> bool


    (*  This tests if the handler will take all of the
        rest of the URL path if there are no child nodes.
    *)
    val canTakeRest:    Config.NodeConfig -> bool

end
```

Communication between the backbone and handler threads is done through a mailbox, which has unlimited buffering of messages. This ensures that a slow handler can't cause congestion by having messages back-up into the tree of backbone threads. The mailbox is created by the handler thread and returned to the backbone thread from the `init` function. Here are the types for a message to a handler and the reply. The protocol is described in more detail below.

```
datatype HndMsg = HndReq of {
            factory:    NodeCreator,
            config:     Config.NodeConfig,
            options:    Options,
            request:    HTTPMsg.Request,
            segs:       string list,    (* remaining path segments *)
```

```
            rchan:      HndReply CML.chan
            }

and HndReply =
        HndResponse of HndMsg * HTTPMsg.Response
    |   HndSprout   of HndMsg * Node
```

Directory nodes can have extra configuration parameters supplied in a `.swerve` file in the directory (see the section called *The Node Parameters* in Chapter 8). Typically these are authorisation parameters. If the `init` function for a directory node returns a `SwerveConfig` record then the generic code will incorporate it into its node information.

Here is the generic `create` function. (The "H" structure is an abbreviation for the `Handler` argument to the functor).

```
fun create {name, arg, config, factory, options, children} =
let
    val Cfg.NodeConfig {auth, ...} = config

    val in_mbox = M.mailbox()
    val node    = Node {name = name, in_mbox = in_mbox}

    val (h_mbox, h_config_opt) = H.init arg

    (*  Update the options and authorisation from the node
        handler.
    *)
    val (final_auth, final_options) =
        case h_config_opt of
          NONE     => (auth, options)
        | SOME cfg => merge_config cfg auth options

    val impl = NodeImpl {
                    name     = name,
                    in_mbox  = in_mbox,
                    hnd_mbox = h_mbox,
                    hnd_reply= CML.channel(),
                    config   = config,
                    auth     = final_auth,
                    options  = final_options,
                    factory  = factory
                    }
```

```
    val gstate = GenState {
                    children = children
                    }
in
    CML.spawn (node_server impl gstate);
    SOME node
end
handle x => (Log.logExn x; NONE)
```

The function produces three results. There is static data about a node in the `NodeImpl` record.

```
datatype NodeImpl = NodeImpl of {
    name:           string,
    in_mbox:        NodeMsg M.mbox,
    hnd_mbox:       HndMbox,            (* msgs to the handler *)
    hnd_reply:      HndReply CML.chan,  (* replies from the handler *)
    config:         Cfg.NodeConfig,     (* the original *)
    auth:           Cfg.NodeAuth,
    options:        Node.Options,
    factory:        Node.NodeCreator
    }
```

The `options` and `auth` fields may differ from the original configuration if the contents of a `.swerve` file was merged in.

The second result is the dynamic state for a node. At the moment this only contains the list of children for a node. Since directory nodes create sub-directory nodes on demand the list of children may change.

```
and GenState = GenState of {
    children:       Node list
    }
```

The third result is the `Node` value itself which is the input interface to the node. The static and dynamic node data are retained by the backbone thread (running the `node_server` function). The `Node` value is returned to the caller.

The server for the backbone thread is a simple dispatcher. It received messages from the parent node through its input mailbox and also receives replies from the handler thread.

```
and node_server impl gstate () =
let
    val NodeImpl {in_mbox, hnd_reply, ...} = impl

    fun loop (state: GenState) =
    let
        val new_state =
            CML.select[
                CML.wrap(M.recvEvt in_mbox,
                    MyProfile.timeIt "GenNode request"
                        (handle_request impl state)),

                CML.wrap(CML.recvEvt hnd_reply, handler_reply impl state)
                ]
    in
        loop new_state
    end
in
    loop gstate
end
```

Here is the body of the function for handling a request from the parent node.

```
and handle_request impl gstate in_msg : GenState =
let
    val HTTPRequest {request, segs} = in_msg

    val NodeImpl {name, config, factory, options,
                    hnd_mbox, hnd_reply, auth, ...} = impl

... omitted material ...
in
    NodeAuth.checkAuth auth request;    (* raises Respond on error *)

    case segs of
      []          => handle_it()
    | (key::rest) => forward_child key rest;

    gstate
end
handle
  Respond resp =>
    let
        val HTTPRequest {request, ...} = in_msg
    in
        U.return request resp;
```

```
        gstate
    end

| x =>
    let
        val HTTPRequest {request, ...} = in_msg
    in
        Log.logExn x;
        U.return request (U.mkServerFail());
        gstate
    end
```

The authorisation is always checked first. If this fails then a `Node.Respond` exception will be raised. This contains a HTTP response describing the failure. The exception handler passes the response back to the HTTP protocol handler using the `ResponseUtils.return` utility function.

If the request is authorised then it is time to see if it is destined for this node or a child node. If the list of remaining segments is empty then the request has reached its target node. The `handle_it` function passes the request to the handler thread.

```
fun handle_it() =
let
    val () = Log.testInform G.TestStoreProto Log.Debug
                (fn() => TF.L ["Node ", name, " handles it"])

    val msg = HndReq {
                factory     = factory,
                config      = config,
                options     = options,
                request     = request,
                segs        = segs,
                rchan       = hnd_reply
                }
in
    M.send(hnd_mbox, msg)
end
```

If the segment list is not empty then it may be destined for a child node. This is dealt with in the `forward_child` function.

```
and forward_child key rest =
```

```
let
    val () = Log.testInform G.TestStoreProto Log.Debug
        (fn() => TF.L [
            "Forwarding to child for key ", key,
            ", rest=", Cfg.listToString rest])

    val GenState {children, ...}  = gstate

    fun match (Node {name, ...}) = (name = key)
in
    case List.find match children of
      NONE       => no_child key rest
    | SOME child => pass_to child request rest
end



and no_child key rest =
let
    val () = Log.testInform G.TestStoreProto Log.Debug
        (fn() => TF.L [
            "No child for key ", key,
            ", rest=", Cfg.listToString rest])
in
    if H.canTakeRest config orelse
        (null rest andalso H.canTakeLast config)
    then
        handle_it()
    else
        U.return request (U.mkNotFound())
end
```

If the head of the list matches the name of a child node then the request will be routed down to the child node along with the tail of the segment list.

If the request doesn't match a child node then it could be that the handler wants to trap it anyway. For example if the URL path is "/a/b" and "/a" is a directory then "b" may be a file in the directory. I don't create distinct resource nodes for each file. Instead the directory node handles all of the files in the directory. Directory handlers trap file names by returning true from the `canTakeLast` function. Other kinds of nodes may want to perform their own intepretation of the trailing part of a path, in which case they will return true from the `canTakeRest` function. The directory node also does

this so that it can follow sub-directories. If neither a child node nor the handler wants the message then a "404 Not Found" response is generated.

Note carefully that an existing child node takes precedence when routing. A URL path will only be passed to a handler if there is no child node that could take it. The directory node handler in the section called *The Directory Node Handler* expects this. It may decide to create a new child node to implement a sub-directory. Subsequent requests for this sub-directory must be routed through to the child node.

Replies from the handler thread are dealt with in the `handler_reply` function.

```
and handler_reply impl gstate reply : GenState =
let
    val NodeImpl {name, config, factory, options, ...} = impl
in
    case reply of
      HndResponse (h_req, resp) =>
        let
            val HndReq {request, ...} = h_req
        in
            U.return request resp;
            gstate
        end

    | HndSprout (h_req, child) =>
        let
            val HndReq {request, segs, ...} = h_req

            val GenState {children} = gstate

            val new_gstate = GenState {
                    children = child::children
                    }

            val rest = if null segs then [] else tl segs
        in
            pass_to child request rest;
            new_gstate
        end

end
```

A reply from a handler could be either a response to a HTTP request or a request to sprout a new child node. HTTP responses are shipped off immediately to the HTTP protocol handler.

The directory node handler creates new child nodes for sub-directories on demand. These have to be added to the resource node tree. The `HndSprout` reply from the handler tells the backbone thread to add the child to the list of children for the node. The original HTTP request is then passed along to the new child node. Passing a HTTP request to a child node is just a matter of re-packaging.

```
and pass_to node req rest : unit =
let
    val new_msg = HTTPRequest {
                    request = req,
                    segs    = rest
                    }
in
    Node.send node new_msg
end
```

The Directory Node Handler

This implements resources that map to regular files in directories. If the `WithSubDirs` option is enabled it effectively mounts a directory tree onto the URL resource tree.

The story starts with some types.

```
datatype State = State of {
        dir:    string      (* the directory path *)
        }

type CreateArg  = string    (* the directory path *)


(*  We can take the rest in order to try to create a chain of
    child directories.
*)
fun canTakeRest _ = true
```

```
fun canTakeLast _ = true
```

I've included the directory's disk path into a state type although it is really static. It avoids having a separate type and value being passed through the rest of the code. The `CreateArg` type is required by the module interface. It is the type for the disk path argument to the `create` function used by the node factory (in the section called *The Node Factory*). The `canTakeRest` and `canTakeLast` functions always return true to catch all URL paths that reach the node as explained in the section called *The Generic Node*.

The handler is initialised with the `init` function.

```
fun init dir_path =
let
    val file = Files.appendFile dir_path ".swerve"

    val opt_config =
        if Files.exists file
        then
            Cfg.processNodeFile file
        else
            NONE

    val state = State {dir = dir_path}
    val mbox  = M.mailbox()
in
    CML.spawn (server mbox state);
    (mbox, opt_config)
end
```

This reads the `.swerve` file in the directory, if it exists. Then it starts the handler thread. The `.swerve` configuration and a mailbox for the thread are returned to the backbone thread. The thread runs a trivial server function that dispatches incoming messages to the `handle_request` function. Here is the body of the function.

```
and handle_request
        (msg as HndReq {factory, config, options, request, segs, rchan})
        state =
let
    val Cfg.NodeConfig {path = node_path, ...} = config
    val Req.Request  {url, abort, ...} = request
```

```
    val URL.HTTP_URL {path = url_path, ...} = url

... omitted material ...
    (*  If we are not at the end of a path then we can only
        try to sprout a child directory.
    *)
    and do_segs []           = index_dir()
    |   do_segs [file]       = do_file false file
    |   do_segs (file::rest) = do_file true file

in
    do_segs segs;
    state
end
```

The `segs` argument in the message is the list of trailing segments for the URL path. For example if the URL path is "/a/b/c" and the node implements the path "/a" then the segment list will be ["b", "c"]. So if the segment list is empty I have the case of a URL path that leads to the directory with no file name. An index of the directory will be generated by the `index_dir` function. The index could be either the `index.html` file if it exists or a listing of the files in the directory. Indexing is described below.

If the segment list contains one element then it is probably the name of a file in the directory although it could be the name of a sub-directory. If there is more than one element then the first must definitely be the name of a directory. The `do_file` function handles these cases.

```
fun do_file dir_only file =
let
    val State {dir, ...} = state
    val Options {follow_sym, with_subdirs, ...} = options

    val file_path = Files.appendFile dir file

    val () = Log.testInform Globals.TestStoreProto Log.Debug
        (fn() => TF.L ["Looking at file ", file_path]);
in
    if not follow_sym andalso Files.isSym file_path
    then
        bad()
    else
    if not dir_only andalso Files.isReg file_path
```

```
    then
    (
        if Files.readableReg file_path
        then
            reply_response(send_file file_path request)
        else
            reply_response(U.mkForbidden())
    )
    else
    if Files.isDir file_path
    then
    (
        if with_subdirs andalso Files.accessibleDir file_path
        then
            let
                val new_node_path = node_path @ [file]
            in
                case sprout_child factory new_node_path file_path of
                  NONE       => reply_response(U.mkServerFail())
                | SOME child => reply_sprout child
            end
        else
            reply_response(U.mkForbidden())
    )
    else
        bad()
end
```

Whether the file argument is a regular file or a directory, its path on disk is built from the disk path of the node's directory. Then we check what kind of file it really is. Symbolic links are followed by default when opening files so, if they are not to be, I must filter them out first. If the path is a regular file and this is reasonable, because there are no more URL segments after the file name and the server has permission to read the file, then I can send it to the client.

If the file is a directory and the `WithSubDirs` configuration option has been specified then the sub-directory must be automatically made into a new node in the resource store. The new node is created by the `sprout_child` function and then sent back to the backbone thread to be inserted into the resource tree. Then the backbone thread will route the original request to the new node and the story starts all over again.

```
and sprout_child factory node_path dir_path : Node.Node option =
let
    val options = Options {
        exec_cgi        = false,
        follow_sym      = false,
        with_subdirs    = true
        }

    val child_config = Cfg.NodeConfig {
            path    = node_path,
            kind    = Cfg.NodeIsDir {path = dir_path},
            options = [],           (* passed directly to the factory *)
            auth    = Cfg.NodeNoAuth
            }
in
    factory {
        config    = child_config,
        children  = [],
        options   = options
        }
end
```

The `sprout_child` function is a wrapper around a call to the node factory. The sub-directory doesn't have an entry in the server configuration otherwise the node would already exist in the tree and there would be no sprouting. So I have to synthesise a configuration that describes a directory node. At the moment I am not doing any inheritance of options in the synthetic configuration except that the `WithSubDirs` option must be true all the way down to get the entire tree under the sub-directory. Although I've specified `NodeNoAuth` for the authorisation, if the sub-directory has a `.swerve` file that specifies some authorisation control it will be merged in when the new node is created (see the `create` function in the section called *The Generic Node*).

Here is the `send_file` function that returns a regular file to the client.

```
and send_file file_path req : Req.Response =
let
    val () = Log.testInform Globals.TestStoreProto Log.Debug
        (fn() => TF.L ["dir_node sends file ", file_path]);

    fun file_response() =
```

```
    let
        val entity = Entity.Entity {
                    info = Entity.emptyInfo,
                    body = Entity.fileProducer file_path
                    }
    in
        Req.Response {
            status  = Status.OK,
            headers = [],
            entity  = entity
            }
    end

in
    if Files.readableReg file_path
    then
        file_response()
    else
        U.mkForbidden()
end
```

All it has to do is create an entity that represents the file on disk and wrap it into a HTTP response record. The `Entity.fileProducer` function makes a producer that can deliver from disk. The producer will fill in the file length and last modification date so I don't have to do it here.

The indexing of directories is controlled by the `index_dir` function mentioned above.

```
and index_dir() =
let
    val State {dir, ...} = state
    val Options {follow_sym, ...} = options
    val Cfg.ServerConfig {dir_index, ...} = Cfg.getServerConfig()

    val file_path = Files.appendFile dir dir_index

    val () = Log.testInform Globals.TestStoreProto Log.Debug
        (fn() => TF.L ["Indexing directory ", file_path]);
in
    if not follow_sym andalso Files.isSym file_path
    then
        bad()
    else
    if Files.isReg file_path
```

```
    then
        reply_response(send_file file_path request)
    else
        reply_response(fancy_index abort url dir)
end
```

The function looks to see if there is an `index.html` file in the directory. (The name `index.html` actually comes from the server configuration). If the file exists and is readable then it is returned. If the file does not exist then the contents of the directory is listed and formatted as HTML and returned (again if accessible). The result is similar to Netscape's directory indexing. This is done by the `fancy_index` function.

```
and fancy_index abort url dir : Req.Response =
let
    val URL.HTTP_URL {host, port, userinfo, path = url_path, ...} = url
    val URL.URLPath  {segs, absolute} = url_path


    fun build entries =
    let
        val text = TF.C [header(), translate entries, trailer()]
    in
        U.mkHTML Status.OK text
    end

... omitted material ...
in
    Log.testInform Globals.TestStoreProto Log.Debug (fn() => TF.L [
                "dir_node accessibleDir of ", dir,
                " is ", Bool.toString(Files.accessibleDir dir)
                ]);

    if Files.accessibleDir dir
    then
    (
        (build(FileIO.listDir abort dir))
            handle _ => U.mkServerFail()
    )
    else
        U.mkForbidden()
end
```

The listing of the directory requires reading from a file descriptor so it must go through the Open File Manager and may be aborted by a time-out. The building of the HTML is a messy bit of text formatting using the `TextFrag` module. The code assumes that there is a "/icons" URL path in the server to fetch icons from. I'll omit the gory details.

The CGI Node Handler

This module handles requests that run CGI scripts. It conforms fairly closely to the CGI version 1.1 specification. The differences are

- The REMOTE_HOST environment variable is not set. This would require a DNS reverse lookup even if the CGI script is not interested. Instead only REMOTE_ADDR is supplied. This actually conforms to the specification but may be unusual.
- The authorisation type will not be accurate unless it is set directly on the CGI node by the node's configuration. Authorisation inherited from higher nodes is not reported. This is a design problem.
- The command line is never set. It would only be used for ISINDEX queries which the server does not support.

The CGI interface has been tested with some simple Perl scripts using the CGI.pm module.

The initialisation of the node is similar to that of the directory node in the section called *The Directory Node Handler* so I won't repeat it here. Instead the story starts with the `handle_request` function.

```
and handle_request
        (msg as HndReq {config, rchan, request, segs, ...})
        script =
let
    val env  = build_environ config request (length segs)
    val resp = run_script script env request
in
    CML.send(rchan, HndResponse(msg, resp))
```

```
end
```

This is simple enough: build the set of environment variables; run the script and send back the response. The `build_environ` function is large and I'll describe it in pieces.

```
and build_environ config request num_left =
let
    val Cfg.NodeConfig {auth, ...} = config

    val Req.Request {url, headers, method, protocol, client, ...}
                                                    = request
    val URL.HTTP_URL {path, query, fragment, ...} = url
    val URL.URLPath {segs, ...} = path

    val script_path = URL.URLPath {
                        segs = List.take(segs, length segs - num_left),
                        absolute = false
                        }

    val trail_path = URL.URLPath {
                        segs = List.drop(segs, length segs - num_left),
                        absolute = false
                        }

    (*  Copy across approved variables.
    *)
    fun copy n =
    (
        case OS.Process.getEnv n of
          NONE   => NONE
        | SOME v => SOME(concat[n, "=", v])
    )

    val copied = List.mapPartial copy
                   ["PATH", "HOSTNAME", "LANG", "LOGNAME",
                    "HOME", "LD_LIBRARY_PATH", "SHELL"]
```

This first section unpacks the arguments and copies variables out of the server's environment. Only those variables that are likely to be useful to a script and that are reasonably safe are copied. The `mapPartial` function suppresses variables that aren't set in the server's environment. The result is a list of strings of the form "name=value".

This next section adds in the unconditional CGI variables. (The ^ is the infix string concatenation operator).

```
val Cfg.ServerConfig {server_name, listen_port, ...} =
        Cfg.getServerConfig()

val basics = [
        "SERVER_NAME="     ^ server_name,
        "SERVER_PORT="     ^ (Int.toString listen_port),
        "SERVER_SOFTWARE=" ^ Globals.cgi_version,
        "REQUEST_METHOD="  ^ (Req.methodToString method),
        "SERVER_PROTOCOL=" ^ protocol,
        "GATEWAY_INTERFACE=CGI/1.1",
        "PATH_INFO="       ^ (URL.pathToString trail_path),
        "SCRIPT_NAME="     ^ (URL.pathToString script_path),

        (*  We don't set REMOTE_HOST, the script can find
            it if it wants.
        *)
        "REMOTE_ADDR="     ^ (NetHostDB.toString client)
        ]
```

The next section builds the optional variables. Each value is a list containing a single "name=value" string. The list is empty if the variable is not being set. The variables can then be easily merged by concatenating the lists.

```
val auth_env : string list =
        case auth of
          Cfg.NodeNoAuth  => []
        | Cfg.NodeBasic _ => ["AUTH_TYPE=Basic"]

val user_env : string list =
        case Hdr.getAuth headers of
          NONE => []
        | SOME (Hdr.AuthBasic (opt_id, pwd)) =>
        (
            case opt_id of
              NONE => []
            | SOME id => ["REMOTE_USER=" ^ id]
        )

val ctype_env : string list =
        case Hdr.getContentType headers of
          NONE       => []
        | SOME mtype => ["CONTENT_TYPE=" ^
```

```
                (TF.toString TF.UseLf (E.formatType mtype))]

val clen_env : string list =
        case Hdr.getContentLength headers of
          NONE     => []
        | SOME len => ["CONTENT_LENGTH=" ^ (Int.toString len)]

val query_env : string list =
        case query of
          NONE   => []
        | SOME q => ["QUERY_STRING=" ^ q]
```

Next all request headers that haven't been covered must be translated to CGI variables. The translation converts a header name such as "User-Agent" to the variable name HTTP_USER_AGENT. The header has to be reconstituted as a string to get the name. Finally the headers are joined together to build the complete list.

```
fun hdr_copy (Hdr.HdrAuthorization _) = NONE
|   hdr_copy (Hdr.HdrConLen _)        = NONE
|   hdr_copy (Hdr.HdrConType _)       = NONE
|   hdr_copy (Hdr.HdrChallenge _)     = NONE
|   hdr_copy (Hdr.HdrBad _)           = NONE
|   hdr_copy header =
let
    (*  Find the initial colon, split off any white space after it.
        Header names become uppercase with hyphens mapped to
        underscores.
    *)
    val text = SS.all(TF.toString TF.UseLf (Hdr.formatHeader header))
    val (left, right) = SS.splitl (isntVal #":") text

    fun cvt #"-" = "_"
    |   cvt c    = str(Char.toUpper c)

    val ename  = SS.translate cvt left
    val evalue = SS.dropl Char.isSpace (SS.triml 1 right)
in
    SOME(concat["HTTP_", ename, "=", SS.string evalue])
end

val other_headers = List.mapPartial hdr_copy headers

val final_headers = List.concat[copied, basics, auth_env,
                      user_env, ctype_env, clen_env, query_env,
```

```
                    other_headers]
```

Here is the run_script function.

```
and run_script script env request : Req.Response =
let
    (*  The Aborted exception can be raised in here. *)
    val Req.Request {abort, ...} = request


    fun talk holder =
    let
        val (proc, _) = ExecReader.get holder
        val ()        = send_entity abort proc request
        val headers   = get_headers abort proc script

        (*  We don't pass these to the client.
            The last four are handled by the Entity Info.
        *)
        fun select (Hdr.HdrStatus _)        = false
        |   select (Hdr.HdrConType _)       = false
        |   select (Hdr.HdrConLen _)        = false
        |   select (Hdr.HdrConEnc _)        = false
        |   select (Hdr.HdrLastModified _)  = false
        |   select _ = true


        val status =
            case Hdr.getStatus headers of
              NONE   => Status.OK
            | SOME s => s


        (*  This includes error responses from the script.
        *)
        fun normal_response() =
        let
            val () = Log.testInform Globals.TestCGIProto Log.Debug
                            (fn()=>TF.S "CGI normal_response")

            val entity = Entity.Entity {
                            info = Hdr.toEntityInfo headers,
                            body = Entity.procProducer holder
                            }
        in
            Req.Response {
```

```
                status  = status,
                headers = List.filter select headers,
                entity  = entity
                }
        end
    in
        normal_response()
    end
    handle _ =>
        (
            kill (#1(ExecReader.get holder));
            U.mkServerFail()        (* REVISIT - should be ReqTimeout *)
        )

in
    (*  The holder will be closed in procProducer after the response
        body has been delivered.  If there is an error then the
        holder will eventually be finalised.
    *)
    case ExecReader.openIt abort (script, [], env) of
      NONE        => U.mkServerFail()  (* error already reported *)
    | SOME holder => talk holder
end
```

The forking and execing of the script is handled by the `ExecReader` module which is described in the section called *The Open File Manager*. This module waits for enough file descriptors before proceeding. It provides for finalisation to kill and reap the child if there is a time-out.

If the script is successfully started then the `talk` function sends any entity body to the `stdin` of the script. Then it reads the headers that come back from the script on `stdout` and constructs a normal response. (The `normal_response` function is a left-over of more complex code that I simplified). The status and entity-specific headers are separated out. An `Entity` value is constructed to represent the body that may or may not be still waiting on `stdout` to be read. The body won't be read until the response is being written to the socket of the connection, as described in the section called *The Connection Protocol* in Chapter 8. The `get_headers` function can raise the local `Aborted` exception if it detects an abort condition. I make an attempt to ensure that the child process is killed quickly rather than wait for finalisation.

Here is the `send_entity` function.

```
and send_entity abort proc request =
let
    val Req.Request {entity, ...} = request
    val (_, ostream) = Unix.streamsOf proc

    val consumer = CML.channel()

    val () = Log.testInform Globals.TestCGIProto Log.Debug
                (fn()=>TF.S "CGI send_entity")

    fun send_it() =
    (
        case CML.recv consumer of
          E.XferInfo _ => send_it()

        | E.XferBytes vec =>
            (
                TextIO.output(ostream, Byte.bytesToString vec);
                send_it()
            )

        | E.XferDone  => done()
        | E.XferAbort => done()
    )

    and done() =
    (
        TextIO.closeOut ostream
    )
in
    E.startProducer abort entity consumer;
    CML.spawn send_it;
    ()
end
```

The sending has to be done in a separate thread because there is no guarantee that the CGI script will even read its `stdin` let alone consume it all strictly before attempting to write to `stdout`. If the script doesn't read its stdin then the sending thread will block indefinitely and will eventually be caught by the garbage collector after the child process has been reaped and all files closed. To send the entity the thread acts as a consumer of the transfer procotol.

Here is the `get_headers` function.

```
and get_headers abort proc script =
let
    val (istream, _) = Unix.streamsOf proc
    val () = Log.testInform Globals.TestCGIProto Log.Debug
                (fn()=>TF.S "CGI get_headers")

    (*  This must match Connect.readLine.
        We strip the terminating \r\n.
    *)
    fun readLine() =
    (
        if Abort.aborted abort
        then
            NONE
        else
        (
            case TextIO.inputLine istream of
              ""   => NONE

            | line =>
                let
                    val l = size line
                in
                    if l > 1 andalso String.sub(line, l-2) = #"\r"
                    then
                        SOME(String.substring(line, 0, l-2))
                    else
                    if l > 0 andalso String.sub(line, l-1) = #"\n"
                    then
                        SOME(String.substring(line, 0, l-1))
                    else
                        SOME line
                end
        )
    )


    (*  Log any bad headers and discard them. *)
    fun check [] out = out
    |   check ((Hdr.HdrBad h)::rest) out =
    (
        Log.error ["CGI ", script, " returned bad header: ", h];
        check rest out
    )
    |   check (h::rest) out = check rest (h::out)
```

```
    (*  Try to read some headers. This will return early on
        an abort.
    *)
    val headers = Hdr.readAllHeaders readLine
in
    if Abort.aborted abort
    then
        raise Aborted
    else
        check headers []
end
```

This is mainly a wrapper around the common `Hdr.readAllHeaders` function. The messy bit is emulating the handling of CR-LF that the `Connect` module does. If a time-out happens while the script is running then it is most likely to be detected while waiting for the headers. I check for an abort condition before each header line and after the headers have been read. The `Aborted` exception breaks out of the `run_script` function.

The Builtin Node Handler

This module implements some simple built-in kinds of nodes. They are used for testing. If you were to use the server as a front-end for an SML application then the interface between the server and the application would be modeled on this module.

The code in this module is fairly generic. It consists of a framework for running a function that creates the response to a request. This is shown here.

```
and handle_request
        (msg as HndReq {config, rchan, request, ...})
        =
let
    val Cfg.NodeConfig {kind, ...} = config

    fun reply response =
    (
```

```
        CML.send(rchan, HndResponse(msg, response))
    )
in
    case kind of
      Cfg.NodeIsBuiltin {name} =>
    (
        case get_maker name of
          NONE   => reply (U.mkServerFail())
        | SOME f => reply (f request)
    )

    | _ => raise InternalError "SimpleBuiltin,handleRequest"
end



and get_maker name =
(
    case name of
      "hw"      => SOME (fn _ => U.mkHelloWorld())
    | "reject"  => SOME (fn _ => U.mkNotFound())
    | "sleep"   => SOME sleep
    | _         => NONE
)
```

The `get_maker` function selects a response-building function depending on the kind of the built-in node as specified in the node's configuration. The hello world and reject nodes return fixed responses. The sleep node delays for a number of seconds specified by the value in the query. If you configure a node as follows:

```
Node /sleep
{
    #  Pass a timeout as a query e.g. /sleep?3
    BuiltIn = "sleep";
}
```

then the URL `http://.../sleep?3` will return a response 3 seconds later. Here is the `sleep` function.

```
and sleep request =
let
    val Req.Request  {url, abort, ...} = request
    val URL.HTTP_URL {query, ...} = url
```

```
    val timeout =
        case query of
          NONE => 1
        | SOME q => getOpt(Int.fromString q, 1)

    val t_evt = CML.timeOutEvt(Time.fromSeconds(Int.toLarge timeout))
in
    CML.select[
        CML.wrap(t_evt, fn _ => ()),
        CML.wrap(Abort.evt abort, fn _ => ())
        ];

    U.mkHTML Status.OK (TF.L [
             "<html><body><p>",
             "Slept for ", Int.toString timeout, " seconds",
             "</body></html>"])
end
```

First it gets the time-out from the query string or defaults to 1 second if it isn't available or readable. Then it uses a `CML.select` to wait for the desired time-out. This must also abort on a request time-out. The `mkHTML` function builds a simple response containing some HTML. See the section called *The ResponseUtils Module*. Note that this handler is single-threaded. So if two requests come in at the same time the second one will start its delay after the first one has finished. The handler should process these requests concurrently but I only use this function for testing at the moment so I'm not bothered.

The ResponseUtils Module

This module contains a collection of miscellaneous functions, mainly for creating HTTP responses. Here's a simple one that returns plain text. It uses a `textProducer` to deliver the entity body out of a string in memory. TF is the TextFrag module described in the section called *The Text Module*.

```
and mkHelloWorld() : Req.Response =
let
    val info = Entity.Info {
```

```
                etype       = SOME (Entity.simpleType "text" "plain"),
                encoding    = NONE,
                length      = NONE,
                last_mod    = NONE
                }

    val entity = Entity.Entity {
                info = info,
                body = Entity.textProducer(TF.C
                            [TF.S "hello world", TF.Nl])
                }
in
    Req.Response {
        status  = Status.OK,
        headers = [],
        entity  = entity
        }
end
```

This next one has a little help to generate HTML.

```
and mkForbidden() =
(
    mkHTML Status.Forbidden
        (lines_to_text [
            "<html><body>",
            "<em>Access Denied</em>",
            "</body></html>"
            ])
)


and lines_to_text lst =
(
    TF.C(map (fn l => TF.C [TF.S l, TF.Nl]) lst)
)



and mkHTML status frag =
let
    val info = Entity.Info {
                etype       = SOME (Entity.simpleType "text" "html"),
                encoding    = NONE,
                length      = NONE,
                last_mod    = NONE
```

```
                }

    val entity = Entity.Entity {
                info = info,
                body = Entity.textProducer frag
                }
in
    Req.Response {
        status  = status,
        headers = [],
        entity  = entity
        }
end
```

The NodeAuth Module

This module checks the client's credentials. It only uses the Basic authorisation type. The implementation is quite simple-minded. The user name and password are looked up each time by reading through the authorisation files. This could be done more efficiently by caching the file contents in memory. But then I would have to have some control mechanism to reload the cache if I change a password or add a user.

The interface is a single function.

```
fun checkAuth auth (req: Req.Request) =
let
in
    case auth of
      Cfg.NodeNoAuth    => ()       (* pass *)
    | Cfg.NodeBasic au  => validate_basic au req
end
```

If the authorisation fails this function constructs a response and returns it via the `Node.Respond` exception. This is caught in the GenericNodeFn functor. See the section called *The Generic Node*.

Here is the top-level of the Basic validation.

```
and validate_basic
```

```
        (auth as {realm, user_file, group_file, users, groups})
        req
        : unit =
let
    val () = Log.testInform G.TestAuth Log.Debug (fn() => TF.L [
                "Basic auth for realm ", realm])

    val Req.Request {headers, abort, ...} = req


    (*  Generate a challenge response to prompt for a password.  *)
    fun challenge() =
    let
        val () = Log.testInform G.TestAuth Log.Debug (fn() => TF.L [
                    "Returning challenge for realm ", realm])

        val resp = Req.Response {
                status  = Status.UnAuth,
                headers = [Hdr.HdrChallenge(Hdr.ChallBasic realm)],
                entity  = Entity.None
                }
    in
        raise Node.Respond resp
    end


    fun reject() =
    let
        val resp = Req.Response {
                status  = Status.UnAuth,
                headers = [],
                entity  = Entity.None
                }
    in
        raise Node.Respond resp
    end

in
    case Hdr.getAuth headers of
      NONE => challenge()

    | SOME (Hdr.AuthBasic (opt_id, pwd)) =>
    (
        case opt_id of
          NONE    => reject()
        | SOME id => validate_user abort auth reject id pwd
    )
```

```
end
```

To get to this function the node must require (Basic) authorisation. So if the request does not have one then the response will contain a challenge header which will make a browser prompt the user for a password and resend the request. If the request has some authorisation then it must have both a user name and password and these are validated against the files. Since reading files takes time there must be a check for an aborted connection. The `abort` value is passed down through the validation code. The `reject` function raises a "401 UnAuthorized" reponse and is passed along to the validation routines.

Here is the top-level of the user validation.

```
and validate_user
        abort
        {realm, user_file, group_file, users, groups}
        rejecter id pwd
        : unit =
let
    val () = Log.testInform G.TestAuth Log.Debug (fn() => TF.L [
                "Validate user=", id, " for realm=", realm])

    val all_users = add_group_users abort users group_file groups
in
    if List.exists (isVal id) all_users
       andalso validate_pwd abort user_file id pwd
    then
        ()
    else
        rejecter()
end
```

The group names in the authorisation record are expanded by `add_group_users` to a list of user names and added to the user name list. Then if the user name is in this list the password must be checked. (The `isVal` function is in the Common module). I'll skip the `add_group_users` function which is a messy bit of file reading and go on to the `validate_pwd` function. This is a simpler bit of file reading.

```
and validate_pwd abort user_file id pwd : bool =
```

```
let
    val () = Log.testInform G.TestAuth Log.Debug (fn() => TF.L [
                "Validate pwd for user=", id, " pwd=", pwd])

    fun loop lnum strm =
    let
        val line = TextIO.inputLine strm
    in
        if line = ""
        then
            false           (* eof so failed *)
        else
            if check_line line lnum
            then
                true
            else
                loop (lnum+1) strm
    end


    and check_line line lnum =
    let
        val (left, right) =
            SS.splitl (isntVal #":") (SS.all line)

        fun clean s = SS.dropr Char.isSpace (SS.dropl Char.isSpace s)

        (*  Trim off leading and trailing white space from the names. *)
        val user     = SS.string(clean left)
        val password = SS.string(clean(SS.triml 1 right))

        val () = Log.testInform G.TestAuth Log.Debug (fn() => TF.L [
                "Found user=", user, " pwd=", password])
    in
        user = id andalso password = pwd
    end
in
    FileIO.withTextIn abort user_file false (loop 1)
end
```

The `FileIO.withTextIn` function takes care of opening and closing the file including waiting for a file descriptor to be available. It passes a text stream to the `loop` function which reads the lines. The `false` value is a default in case the file could not be read. A line is of the form "user: pwd" with white

space allowed around the user name and the password. The `clean` function trims off this white space.

The IETF Layer

This layer implements modules that deal with the kinds of data that make up internet protocol messages (as described in the RFCs). For the purposes of HTTP, this includes headers, status messages and mime-encoded entities.

The Entity Module

This module implements the `Entity` type for all HTML pages, images etc. that are transferred over HTTP. It also includes a simple bit of MIME type handling. The functions for transferring entities around the server are in here. This includes the transfer protocol (the `XferProto` type) and the producer/consumer system. See the section called *Entities, Producers and Consumers* in Chapter 8 for an overview.

First here is the MIME type interface.

```
datatype MType =
        MType of {
            mtype:      string,
            msubtype:   string,
            mparams:    (string * string) list
            }

    |   MTypeUnknown

val formatType:     MType -> TextFrag.Text

(*  This creates a simple type e.g. text/plain.
*)
val simpleType:     string -> string -> MType

(*  This works out a Mime type for a file. It only
    looks at the file name.
```

```
*)
val getMimeType:    string -> MType
```

This just declares the type and provides some utility functions. The declaration is needed here for the entity info. The parsing of MIME types in headers is taken care of in the HTTPHeader module (see the section called *The HTTPHeader Module*). The `formatType` function converts the type back to the text format suitable for a header. The `getMimeType` function maps the file name extension to a MIME type using the types file specified by the `TypesConfig` configuration parameter (see the section called *The Server Parameters* in Chapter 8). Most of the work for this is done in the Config module in the section called *The Config Module - Interface*. These functions are simple enough to not need further explanation here.

The `Encoding` type is treated similarly to `MType` above. The parsing of encodings is done in the HTTPHeader module. The type is declared here for the entity info. The `formatEncoding` encoding function converts an encoding value back to the text format for a header. I won't discuss if further here. The server never takes notice of the encoding. Entities passing through are never decoded. When the server has to generate an entity such as a message or a fancy index for a directory they are never encoded. I don't recognise compression in disk files.

As explained in the section called *Entities, Producers and Consumers* in Chapter 8 the interface for an entity is abstract. The data is delivered by a producer function. The entity body itself is represented by a function that can create a producer function.

```
datatype Entity =
        Entity of {
            info:   Info,
            body:   MKProducer
            }
    |   None

and Info = Info of {
        etype:      MType option,
        encoding:   Encoding option,
        length:     int option,
```

```
        last_mod:   Date.date option
        }
```

The Info type contains information about the entity. This corresponds to the `Content-Type`, `Content-Encoding`, `Content-Length` and `Last-Modified` HTTP headers. Not all of the Info fields are used by all of the different kinds of producer function. For example if the entity is stored in a disk file then the length and last modification date are taken from the file instead of the Info. The MIME type is derived from the file name extension but the encoding is ignored. I should either look at the extension or try to detect the type of the file from the first few bytes. But ignoring the encoding will be enough for this simple server at the moment.

Here are the interface declarations for the transfer protocol and producer/consumer system. The protocol is described in more detail in the section called *Entities, Producers and Consumers* in Chapter 8.

```
(*  A producer sends messages of this type to its consumer. *)
and XferProto =
        XferInfo  of Info               (* send this first *)
    |   XferBytes of Word8Vector.vector (* then lots of these *)
    |   XferDone                        (* then one of these *)
    |   XferAbort                       (* or else one of these *)
withtype Consumer = XferProto CML.chan
     and MKProducer = Abort.Abort -> Info -> Consumer -> CML.thread_id

(*  This creates a producer for an entity. *)
val startProducer:  Abort.Abort -> Entity -> Consumer -> CML.thread_id

val textProducer:   TextFrag.Text -> MKProducer
val tmpProducer:    TmpFile.TmpFile -> MKProducer
val fileProducer:   string -> MKProducer

(*  Beware that process producers are one-shot.
    The holder is closed after the entity has been produced.
*)
val procProducer:   ExecReader.Holder -> MKProducer
```

To create an `Entity` value which represents a disk file you would use the `fileProducer` function. This returns a `MKProducer` function which in turn

can be used to make multiple concurrent producers. Each producer delivers the contents of the file using the transfer protocol.

The `startProducer` function starts the delivery process.

```
and startProducer abort (Entity {info, body}) consumer = body abort info consumer

|   startProducer abort None consumer =
(
    CML.spawn (fn () => CML.send(consumer, XferDone))
)
```

All it does is call the producer function. It handles the case of a non-existent entity by starting a producer thread that just sends the `XferDone` message.

Here is the `fileProducer` function.

```
and fileProducer name abort old_info consumer =
let
    fun producer() =
    let
        (*  All of the info fields are regenerated from the
            file at the time we send it.
        *)
        val opt_len = FileIO.fileSize name
        val modt = Option.map Date.fromTimeUniv (FileIO.modTime name)

        val info = Info {
                    etype       = SOME(getMimeType name),
                    encoding    = NONE,
                    length      = opt_len,
                    last_mod    = modt
                    }
    in
        CML.send(consumer, XferInfo info);

        case opt_len of     (* NONE if can't access the file *)
          NONE     => CML.send(consumer, XferDone)
        | SOME len => send_file()
    end


    and send_file() =
    let
        (*  Record the open file so that it can be finalised if
            the consumer is aborted e.g. due to a connection timeout.
```

```
        *)
        fun loop strm =
        (
            if Abort.aborted abort
            then
                CML.send(consumer, XferAbort)
            else
            let
                val chunk = BinIO.inputN(strm, file_chunk)
            in
                if Word8Vector.length chunk = 0
                then
                    CML.send(consumer, XferDone)
                else
                (
                    CML.send(consumer, XferBytes chunk);
                    loop strm
                )
            end
        )
    in
        case BinIOReader.openIt abort name of
          NONE   => ()
        | SOME h => (loop (BinIOReader.get h); BinIOReader.closeIt h)
    end
    handle x => (Log.logExn x; ())

in
    CML.spawn producer
end
```

I am using currying here. The function call (`fileProducer "foo.html"`) returns a function that takes abort, Info and consumer arguments and starts the producer thread and returns its id. This returned function has the type `MkProducer`. When all of the arguments to the `fileProducer` function are eventually supplied it spawns a thread which runs its `producer` function. New entity info is derived from the file each time that a producer thread is spawned. This allows changes to the file length and modification time to be noticed. There is no safety check for a file changing as it is being delivered. If this happens then the `Content-Length` header won't match the amount of data actually sent.

If the file is of non-zero length then its contents are sent by the `send_file` function. This does some binary I/O to read the file in chunks and deliver them in `XferBytes` messages. The BinIOReader module takes care of waiting for free file descriptors and closing the file on an abort. (See the section called *The Open File Manager*). I also need to check for the abort condition while sending the file. The CML library has no function like "`inputNEvt`" which returns an event for when data is ready from a `BinIO.instream`. I have to poll for the abort condition before each file read. This is a case where the server may end up trying to send data to an aborted connection. This will be caught when an attempt is made to write to a closed connection socket.

The `tmpProducer` delivers from a temporary file. This is just a particular case of the `fileProducer`. The `textProducer` delivers from a `TextFrag` in memory. The length is obtained from the length of the text and the other Info must be supplied. Each fragment of the text is sent as a separate `XferBytes` message.

```
and textProducer frag abort einfo consumer =
let
    val len = TF.length TF.UseCrLf frag

    fun producer() =
    (
        CML.send(consumer, XferInfo(update_length einfo len));
        TF.apply TF.UseCrLf send frag;
        CML.send(consumer, XferDone)
    )

    and send str = CML.send(consumer, XferBytes(Byte.stringToBytes str))
in
    CML.spawn producer
end
```

The `procProducer` function delivers an entity from a pipe that is reading from a CGI script. This makes it a bit different from the other producers in that it can only work once. The Info for the entity is obtained from the headers returned by the CGI script. See the section called *The CGI Node Handler* for more details.

```
and procProducer (holder: ExecReader.Holder) abort einfo consumer =
let
    val opened as (proc, _) = ExecReader.get holder
    val (strm, _) = Unix.streamsOf proc

    fun producer() =
    (
        CML.send(consumer, XferInfo einfo);
        send_file();
        ExecReader.closeIt holder;
        ()
    )


    and send_file () =
    (
        (*  See send_file above
            PROBLEM: CML timeouts don't seem to interrupt the inputN
            operation.
        *)
        if Abort.aborted abort
        then
        (
            CML.send(consumer, XferAbort)
        )
        else
        let
            val chunk = TextIO.inputN(strm, pipe_chunk)
        in
            if chunk = ""
            then
            (
                CML.send(consumer, XferDone)
            )
            else
            (
                CML.send(consumer, XferBytes(Byte.stringToBytes chunk));
                send_file()
            )
        end
        handle x => (Log.logExn x; ())
    )
in
    CML.spawn producer
end
```

The `producer` function is straight-forward enough: send the info then send the file then close. Sending the file consists of a loop to read chunks from the pipe and deliver them in `XferBytes` messages. As usual I need to check for an abort condition each time around.

The HTTPHeader Module

The HTTPHeader module handles the parsing of the header lines in messages. This isn't rocket science, just a lot of string handling, so I won't go through all of the code in detail. I'll just describe the overall layout.

Here is the type for a header.

```
datatype Header =
        HdrDate of Date.date
    |   HdrPragma of string

    |   HdrAuthorization of Authorization
    |   HdrFrom of string
    |   HdrIfModified of Date.date
    |   HdrReferer of string
    |   HdrUserAgent of string

    |   HdrConEnc of Entity.Encoding(* content encoding *)
    |   HdrConLen of int            (* content length *)
    |   HdrConType of Entity.MType  (* mime type *)
    |   HdrLastModified of Date.date

    |   HdrChallenge of AuthChallenge

    (*  These can appear in CGI script output. *)
    |   HdrStatus of HTTPStatus.Status
    |   HdrLocation of URL.URL

    |   HdrExt of (string * string) (* extensions *)
    |   HdrBad of string            (* unparsable junk *)


and Authorization =
        AuthBasic of (string option * string)  (* user id and password *)

and AuthChallenge =
```

```
        ChallBasic of string    (* the realm *)
```

The well-known headers are separated out. Anything that isn't recognised is thrown into the extension category (`HdrExt`) and left as a pair of strings for the header name and value. The status header is included since it appears in the CGI protocol. (See the section called *The CGI Node Handler*). Any header that cannot be parsed is thrown into the bad category (`HdrBad`) for later error reporting.

Here is the header interface.

```
val readAllHeaders: (unit -> string option) -> Header list

val parseHeader:    string -> Header

val formatHeader: Header -> TextFrag.Text

(*  These functions retrieve well-known headers. *)

val getContentLength:   Header list -> int option
val getContentType:     Header list -> Entity.MType option
val getContentEnc:      Header list -> Entity.Encoding option
val getContentMod:      Header list -> Date.date option
val getAuth:            Header list -> Authorization option
val getStatus:          Header list -> HTTPStatus.Status option
val getLocation:        Header list -> URL.URL option

(*  This extracts the relevant headers to build the entity info
    record.
*)
val toEntityInfo:       Header list -> Entity.Info

(*  This overrides one set of headers with another. *)
val overrideHeaders:    Header list -> Header list -> Header list

(*  This excludes a set of headers.  The excluded set is
    demonstrated by sample headers in the first list.
*)
val excludeHeaders:     Header list -> Header list -> Header list
```

The `readAllHeaders` function reads and parses all of the header section of a message. It stops after the blank line that ends a header section. The argument is a function for reading lines as strings from a data source. The

lines must have any trailing CR-LF trimmed off. The `Connect.readLine` function matches this requirement.

The `readAllHeaders` function uses `parseHeader` to parse each header. This function can be called separately. The `formatHeader` function restores a header to text form as a `TextFrag`.

Next come a group of utility functions which fetch particular headers from a list. The `toEntityInfo` function extracts those headers relevant to the contents of an entity and builds an Info value (see the section called *The Entity Module*). The override and exclude functions allow merging groups of headers. They aren't actually used anymore.

The `readAllHeaders` function has this general scheme.

```
fun readAllHeaders readLine : Header list =
let
... omitted material ...
    val lines = loop []
    val hdr_lines = merge lines [] []
    val headers = map parseHeader hdr_lines
in
    (* show_lines hdr_lines; *)
    headers
end
```

The `loop` function reads in all of the header lines into a list. The `merge` function merges continuation lines. If a line starts with white space then it is a continuation of the previous line. The leading white space of the continuation line is stripped off. Then each line is parsed.

The `parseHeader` function has more meat in it.

```
and parseHeader line : Header =
let

    val dispatch = [
        ("DATE",                parse_date HdrDate),
        ("PRAGMA",              parse_pragma),
        ("AUTHORIZATION",       parse_auth),
        ("FROM",                parse_from),
        ("IF-MODIFIED-SINCE",   parse_if_modified),
        ("REFERER",             parse_referer),
```

```
        ("USER-AGENT",          parse_useragent),
        ("CONTENT-ENCODING",    parse_cont_encoding),
        ("CONTENT-LENGTH",      parse_cont_length),
        ("CONTENT-TYPE",        parse_cont_type),
        ("LAST-MODIFIED",       parse_date HdrLastModified),
        ("WWW-AUTHENTICATE",    parse_challenge),
        ("LOCATION",            parse_location),
        ("STATUS",              parse_status)
        ]

    (*  The value has the leading and trailing white space removed. *)
    fun parse sstoken ssvalue =
    let
        val value = (SS.string(SS.dropl Char.isSpace
                                  (SS.dropr Char.isSpace ssvalue)))
        val token  = SS.string sstoken
        val utoken = upperCase token
    in
        case List.find (fn (n, _) => n = utoken) dispatch of
          NONE => HdrExt (token, value)

        | SOME (n, f) => f value
    end


    (*  The common characters are caught early for speed. *)
    fun is_token c = Char.isAlphaNum c orelse c = #"-" orelse
                     Char.contains "!#$%&'*+.^_`|~" c orelse
                     (ord c >= 128)


    val (name, rest) = SS.splitl is_token (SS.all line)
in
    (*  Expect a token, colon and more parts. *)
    if not (SS.isEmpty name) andalso SS.sub(rest, 0) = #":"
    then
        parse name (SS.triml 1 rest)
    else
        HdrBad line
end
```

Down the bottom I first separate out the header name which is a "token" in the IETF terminology. Splitting uses the `Substring` type to avoid copying parts of strings. The `parse` function converts the token to upper case and looks it up in the dispatch table. The dispatch functions are passed the value

of the header line after the colon with the leading and trailing white space stripped. These functions must return a `Header` value, possibly `HdrBad`.

The date parsing dispatch function is shared by the different headers. The header's constructor is passed as an argument to the function. Remember that a constructor in a datatype is equivalent to a function that constructs the type. See the `parse_date` function below.

Some header values consist of multiple tokens that need further parsing. This is described in more detail in the section called *The IETF_Line and IETF_Part Modules*. The result is a list of parts that are described by the following type in the IETF_Part module. (In the code I abbreviate `IETF_Part` to `IP` and `IETF_Line` to `IETF`.).

```
datatype Part =
        Token of string     (* including quoted strings *)
    |   TSpec of char
    |   TWh of string       (* the white space *)
    |   TBad  of char       (* invalid character *)
    |   TEOF
```

Here is a simple header parsing function for the Pragma header.

```
and parse_pragma value =
let
    val hparts = IETF.split value
in
    case strip_ws hparts of
      [IP.Token s] =>
    (
        if field_match s "no-cache"
        then
            HdrPragma "no-cache"
        else
            HdrBad value
    )

    | _ => HdrBad value
end
```

It splits the header value into parts and then checks that this results in exactly one token. The only token that is recognised is `no-cache`. The `field_match` function does case-insensitive matching of two strings.

The most complicated parsing function is for dates. There are three different date formats that are allowed in date headers. See the section called *The Date Header* in Chapter 8 for more details. Here is the top-level of the function.

```
and parse_date (constr: Date.date -> Header) value =
let
    val hparts = IETF.split value
... omitted material ...
in
    (* print "looking at the date parts ";
       IETF.dump parts; print "\n"; *)

    case strip_ws hparts of
      [IP.Token wkday,
       IP.TSpec #",",
       IP.Token day,
       IP.Token month,
       IP.Token year,
       IP.Token hh,
       IP.TSpec #":",
       IP.Token mm,
       IP.TSpec #":",
       IP.Token ss,
       IP.Token "GMT"] => build wkday day month year hh mm ss

    | [IP.Token wkday,
       IP.TSpec #",",
       IP.Token dmy,        (* hyphen isn't special *)
       IP.Token hh,
       IP.TSpec #":",
       IP.Token mm,
       IP.TSpec #":",
       IP.Token ss,
       IP.Token "GMT"] =>
        (
            case String.fields (fn c => c = #"-") dmy of
              [day, month, year] =>
                  build wkday day month ("19"^year) hh mm ss

            | _ => HdrBad value
```

```
        )

    | [IP.Token wkday,
       IP.Token month,
       IP.Token day,
       IP.Token hh,
       IP.TSpec #":",
       IP.Token mm,
       IP.TSpec #":",
       IP.Token ss,
       IP.Token year] => build wkday day month year hh mm ss

    | _ => HdrBad value
end
```

The date value is split into tokens and then a big case expression matches it against each of the date formats. The compiler will be able optimise these cases to efficient code. The second format is trickier since the hyphen character is not considered a token separator. The day-month-year field must be split again into fields on the hyphen character. The `build` function assembles the field values into a `Date.date` value. This involves recognising month and weekday names. I'll skip describing that here.

The `formatHeader` function converts each header back to text as fragments. This avoids all of the copying that would result from concatenating strings.

```
and formatHeader (HdrDate date) =
(
    format_date "Date: " date
)

|   formatHeader (HdrPragma pragma) =
(
    TF.L ["Pragma: ", IETF.quoteField pragma]
)
... omitted material ...
```

The `quoteField` function reintroduces quoting for special characters as described in the section called *HTTP Requests* in Chapter 8.

The remaining functions in this module are simple utility functions that need no further explanation.

The IETF_Line and IETF_Part Modules

The IETF_Line module contains the code for splitting a string into tokens and special characters according to the syntax in the section called *HTTP Requests* in Chapter 8. The result is a list of parts described by this type in the IETF_Part module.

```
datatype Part =
        Token of string     (* including quoted strings *)
    |   TSpec of char
    |   TWh of string       (* the white space *)
    |   TBad  of char       (* invalid character *)
    |   TEOF
```

To help recognise the tokens and special characters I've used a lexer generated by the ML-Lex utility (which is part of the SML/NJ distribution). ML-Lex is similar to the standard Unix lex utility for the C language. You provide a specification of regular expressions for the various parts you want to recognise and it builds a lexer for these expressions. Here is the body of the specification from the `ietf.lex` file.

```
%structure IETFLex
%full

ctl=[\000-\031\127];
ws=[\ \t];
tokn=[!#$%&'*+.0-9A-Z^_`a-z|~\h-];
str=[^\000-\031\127"];

%%

{ws}+           => (TWh yytext);
{tokn}+         => (Token yytext);
\"{str}*\"      => (fix_str yytext);

"("             => (TSpec (String.sub(yytext, 0)));
")"             => (TSpec (String.sub(yytext, 0)));
"<"             => (TSpec (String.sub(yytext, 0)));
">"             => (TSpec (String.sub(yytext, 0)));
"@"             => (TSpec (String.sub(yytext, 0)));
","             => (TSpec (String.sub(yytext, 0)));
";"             => (TSpec (String.sub(yytext, 0)));
":"             => (TSpec (String.sub(yytext, 0)));
```

```
"\\"            => (TSpec (String.sub(yytext, 0)));
"\""            => (TSpec (String.sub(yytext, 0)));
"/"             => (TSpec (String.sub(yytext, 0)));
"["             => (TSpec (String.sub(yytext, 0)));
"]"             => (TSpec (String.sub(yytext, 0)));
"?"             => (TSpec (String.sub(yytext, 0)));
"="             => (TSpec (String.sub(yytext, 0)));
"{"             => (TSpec (String.sub(yytext, 0)));
"}"             => (TSpec (String.sub(yytext, 0)));

.               => (TBad (String.sub(yytext, 0)));
```

The generated SML file will contain a structure named `IETFLex`. This contains these declarations (among others).

```
structure IETFLex=
struct
    structure UserDeclarations =
    struct
        open IETF_Part
        type lexresult = Part

        fun eof() = TEOF

        (*      Strip off the surrounding quotes. *)
        fun fix_str s = Token(String.substring(s, 1, size s - 2))
    end

    fun makeLexer yyinput = ...
    ...
end
```

The contents of the `UserDeclarations` structure is copied in from the top part of the `ietf.lex` file. The `lexresult` declaration is required. It gives the type of the part that is returned by the lexer. The right-hand side of a regular expression specification must be an expression of this type. As in C lex, a variable named `yytext` is available containing the matched string. The `eof` function is also required. It will be called at the end of the lexer's input.

The `makeLexer` function returns a lexer function that can be called successively to get each part. So the lexer function is imperative. The

`yyinput` argument to `makeLexer` is a function that the lexer can call to fetch chunks of the input string. It takes an integer argument for the preferred chunk length, which you can ignore if you like. The end of the input is indicated when `yyinput` returns the empty string.

Here is the `IETF_Line.split` function that operates the lexer.

```
fun split str : IP.Part list =
let
    val done = ref false
    fun input n = if !done then "" else (done := true; str)

    val lexer = IETFLex.makeLexer input

    fun read toks =
    (
        case lexer() of
          IP.TEOF => rev toks
        | t       => read (t::toks)
    )
in
    read []
end
```

I pass the string to be split in a single chunk to the lexer. I have to arrange for the second call to the input function to return an empty string. This requires a kludge with a state variable. The imperative nature of the lexer tends to poison like this all code that interacts with it. The `read` function is a simple loop that keeps getting parts from the lexer until the end-of-file part is found. A list of the parts is returned.

Complementing the `split` function is the `join` function. This converts a list of parts back into a string. At the moment this only used by the `quoteField` function. (In an earlier version of the server I used the `join` function in more places).

```
and join hparts =
let
    fun to_str []         rslt = concat(rev rslt)
    |   to_str [IP.TWh _] rslt = to_str [] rslt    (* trailing ws *)
    |   to_str ((IP.Token s1)::r) rslt = to_str r ((quote s1)::rslt)
    |   to_str ((IP.TWh s)::r)    rslt = to_str r (s :: rslt)
```

```
    |   to_str ((IP.TSpec c)::r)  rslt = to_str r ((str c) :: rslt)
    |   to_str ((IP.TBad  c)::r)  rslt = to_str r rslt
    |   to_str (IP.TEOF::r)       rslt = to_str r rslt


    and quote str =
    let
        (* If there are unsafe characters then right won't be empty.
        *)
        val (_, right) = SS.splitl safe (SS.all str)
    in
        if SS.isEmpty right
        then
            str
        else
            strip_dq str
    end


    and safe c = not (Char.isCntrl c orelse
                      Char.isSpace c orelse
                      Char.contains "()<>@,;:\\\"/[]?={}" c)

    and strip_dq str =
    let
        val fields = SS.fields (fn c => c = #"\"") (SS.all str)
    in
        concat("\"" :: ((map SS.string fields) @ ["\""]))
    end

in
    to_str hparts []
end
```

The `to_str` function is the main loop that converts each part to a string, building a list of strings. Then this list is concatenated. Trailing white space is deleted. The text of tokens is quoted if they contain unsafe characters. I use the `Substring.splitl` function as a simple way to search for a character that matches a predicate. If there are any unsafe characters then the whole token is enclosed in double quotes. Since the HTTP v1.0 specification does not allow double quote characters inside quoted strings I just delete them, for want of a better solution. There shouldn't be any of them inside tokens anyway.

Now I can implement the `quoteField` function as just a split followed by a join.

```
and quoteField field = join(split field)
```

The HTTPStatus Module

This module provides a simple abstraction for status codes. The codes are classified by severity and protocol version. The text description of the code can be generated.

There's not much to say about this. Each code is made into an exported value. The type is abstract.

```
datatype Severity =
    Info | Success | Redirect | ClientError | ServerError

type Status

val OK:             Status          (* 200 *)
val Created:        Status          (* 201 *)
val Accepted:       Status          (* 202 *)
val NoContent:      Status          (* 204 *)
... omitted material ...
val formatStatus:   Status -> string

val severity:       Status -> Severity
val code:           Status -> int

val isV11:          Status -> bool
val same:           Status -> Status -> bool

val fromInt:        int -> Status

(*  This tests if the response needs a body according to the
    status code.  See section 7.2 of RFC1945.
*)
val needsBody:      Status -> bool
```

The HTTPMsg Module

This module defines types for the `Request` and `Response` types that pass between the HTTP protocol section and the resource store. For more information see the section called *Requests and Responses* in Chapter 8.

The Config Layer

This layer implements modules that parse the server's configuration files. This includes the MIME types file. It does not include the authorisation files for user names and passwords. See the section called *The NodeAuth Module* for more information on those.

The configuration file is complex enough that I've used an ML-Yacc parser. The ML-Yacc system is simple enough to use that it can be comfortably used for small jobs like this. The ConfigTypes module defines the types for the parse tree for the main configuration file.

Most of the code is in the Config module. I'll describe that first. Most of it is just a lot of string handling and checking of parameters for correctness so I'll skim lightly over that.

The Config Module - Interface

First here are the types that describe the server's configuration.

```
(*  This is a simplified form of URLPath with just the parts.
    These paths are case-sensitive and so are stored in the
    original case.
*)
type NodePath = string list


(*  Required parameters are stored as strings with "" for an undefined
    value.  Optional ones as string option.
*)
```

```
datatype ServerConfig = ServerConfig of {
            server_root:    string,
            config_dir:     string,
            var_dir:        string,
            tmp_dir:        string,
            doc_dir:        string,
            cgi_dir:        string,
            mime_file:      string,
            error_log:      string,
            dir_index:      string,

            log_level:      Log.Level,

            run_user:       string option,
            run_group:      string option,

            conn_timeout:   int option,
            max_clients:    int option,
            max_tmp_space:  int option,
            max_req_size:   int option,

            listen_host:    string option,
            listen_port:    int,
            server_name:    string
            }
```

The server configuration is described by the `ServerConfig` type. It is held in a static variable in the Config module and fetched by the `getServerConfig` function. It is just a big record of all of the parameters that apply to the server as a whole, as described in the section called *The Server Parameters* in Chapter 8.

A node configuration is described by the `NodeConfig` type. See the section called *The Node Parameters* in Chapter 8 for more information on these configuration parameters.

```
datatype NodeConfig = NodeConfig of {
            path:       NodePath,
            kind:       NodeKind,
            options:    NodeOptionFormula list,
            auth:       NodeAuth
            }

and NodeKind =
```

```
        NodeIsDir of {
            path:   string          (* directory path *)
            }

    |   NodeIsBuiltin of {
            name:   string
            }

    |   NodeIsScript of {
            path:   string
            }

(*  This is a subset of NodeConfig for .swerve files. *)
and SwerveConfig = SwerveConfig of {
            options:    NodeOptionFormula list,
            auth:       NodeAuth
            }

and NodeOptionFormula =
        NOFInherit
    |   NOFAll
    |   NOFNone
    |   NOFAdd of NodeOption
    |   NOFSub of NodeOption


and NodeOption =
        NodeExecCGI
    |   NodeFollowSymLinks
    |   NodeWithSubDirs


and NodeAuth =
        NodeBasic of {
            realm:      string,
            user_file:  string,     (* path to the user file *)
            group_file: string,     (* path to the group file *)
            users:      string list,(* users to allow *)
            groups:     string list (* groups to allow *)
            }

    |   NodeNoAuth
```

The options for a node are described by a formula. This neatly describes how the options of a node can be computed from those of its parent by

interpreting the formula. (The `NodeExecCGI` option is a left-over from an earlier design. It is not used anymore).

The `NodeAuth` type describes the authorisation parameters in a straight-forward way. Extra kinds of authorisation can be added to this type.

Each directory that implements a node can have a `.swerve` file that provides more parameters, mainly for authorisation. The contents of this file is described by the `SwerveConfig` type. Having a separate type allows parameters unique to this file to be added in the future and besides, the path and kind parameters are not relevant.

Here are the main functions of the module interface for the configuration parameters.

```
val processConfig:      string -> unit

val haveServerConfig:   unit -> bool

(*  This is not defined if the processConfig() has not succeeded. *)
val getServerConfig:    unit -> ServerConfig

(*  Return a list of all of the node configurations.
*)
val getNodeConfigs:     unit -> NodeConfig list

(*  This reads a .swerve file and returns the configuration or
    NONE if it wasn't parsable.
    The Io exception will be raised if the file cannot be read.
*)
val processNodeFile:    string -> SwerveConfig option

(*  This returns the node configuration from the main configuration
    file if the node path appeared exactly in the file.
*)
val findNodeConfig:    NodePath -> NodeConfig option
```

The main entry point is `processConfig` which is called from the Main module as soon as the `-f` command line option is read. This reads the configuration files and saves the information in the static variables. If there are any errors then these are reported on standard error and the server configuration will not be saved.

The `haveServerConfig` function tests that the parsing was successful. If so then some of the parameters are poked into global variables, for example the logging file and level. The `getServerConfig` function can then be called from anywhere in the server to get the configuration. Since this information is immutable it can be safely called from any thread.

The node configurations are stored separately as a list in no particular order. The `getNodeConfigs` function fetches the list. Alternatively the `findNodeConfig` function can be used to get a particular node if its configuration path is known. But the main use for these configuration records is to build the resource store tree (see the section called *The Store Module*) and this uses the whole list.

The `processNodeFile` is used to parse a `.swerve` file. Errors are logged in the usual way. The result is returned if there was no error. Nothing is saved in static variables.

The static variables are managed like this.

```
val cf_server_config: ServerConfig option ref = ref NONE

val cf_nodes: NodeConfig list ref = ref []

fun getServerConfig()  = valOf(!cf_server_config)
fun haveServerConfig() = isSome(!cf_server_config)
fun getNodeConfigs()   = !cf_nodes


fun getServerRoot() =
let
    val ServerConfig{server_root, ...} = getServerConfig()
in
    server_root
end
```

The `processConfig` function looks like this.

```
fun processConfig file : unit =
let
    (* show the warnings while processing *)
    val _ = Log.lowerLevel Log.Warn
    val sections = parse_config false file
```

```
in
    (* dump_sections sections; *)

    (*  Ensure that the server node is processed first to
        get the server root for the nodes' files.
    *)
    process_server_section sections;
    process_node_sections sections;

    process_mime_file();
    init_globals();
    ()
end
handle _ => (Log.flush(); raise FatalX)



(*  This pokes some config parameters into various modules in
    common.
*)
and init_globals() =
let
    val ServerConfig {error_log, log_level, max_tmp_space, ...} =
            getServerConfig()
in
    (*  Don't change the error stream until the config has been processed. *)
    Log.flush();
    Log.setLogFile error_log;
    Log.setLevel   log_level;

    case max_tmp_space of
      NONE   => ()
    | SOME l => TmpFile.setDiskLimit l;

    ()
end
```

The log level needs to be lowered to make sure that warnings from the configuration checking get through. They will appear on standard error. (The level could have been set higher by a command line option). Only after the configuration has been successfully read are errors redirected to the log file. Any exceptions are fatal.

The next sections describe the use of ML-Yacc and ML-Lex in detail for parsing the configuration.

The Configuration Grammar

The modules of an ML-Yacc parser are quite complex to describe as they use multiple functors to assemble the parser from parts. But it's easy to copy from an example. For the full details see the ML-Yacc documentation that comes with the source package (see Appendix C).

The parts of the parser are:

- A lexer generated by ML-Lex. This splits the input file into lexical tokens.
- The parsing tables generated by ML-Yacc. This includes modules that define the types and values for the lexical tokens. These are used by the lexer.
- The parsing algorithm which is part of the ML-Yacc library.

There is a lot of superficial similarity with a grammar file for standard Unix C yacc. One big difference is that it is even more strongly recommended that the parsing be side-effect free. I've seen many people write yacc grammars with action code that goes updating data structures or printing out error messages during the parsing operation. This immediately clashes with any kind of back-tracking such as error recovery. The well-known problems of putting action code in the middle of a production are an example of this.

I always just use a parser to build a parse tree. I don't report semantic errors until a later pass over the tree. The ML-Yacc parser here does the same. The action code attached to each production is just an expression that builds a node (or fragment of a node) in the parse tree. The ML-Yacc parser attempts recovery from syntax errors to continue parsing for as long as possible. This works best if the action code has no side-effects.

Here is the top part of the grammar file. It is structured like a the C yacc grammar file.

```
open Common
open ConfigTypes

%%

%eop EOF

(* %pos declares the type of positions for terminals.
   Each symbol has an associated left and right position. *)

%pos Common.SrcPos
%arg (file): string

%term
          KW_SERVER
        | KW_NODE

        | SYM_SEMICOLON
        | SYM_COMMA
        | SYM_LBRACE
        | SYM_RBRACE
        | SYM_EQUALS
        | SYM_SWERVE

        | TOK_WORD of string
        | TOK_STRING of string
        | TOK_INT of int

        | EOF

%nonterm
          start of Section list
        | section_list of Section list
        | section of Section
        | part_list of SectionPart list
        | part of SectionPart
        | literal_list of Literal list
        | literal of Literal

%name Config

%noshift EOF
%pure
```

```
%verbose

%%
```

It starts off with a header containing any SML declarations you may need for the rest of the grammar. This is delimited by the %% characters.

Then comes SML type declarations for the terminals and non-terminals. These must look like an SML datatype declaration. The terminals become the tokens for the lexer. The non-terminals are type declarations for the production rules of the grammar. The action code of the rules must be expressions of these types.

The terminal declarations can carry data. Here for example both a word and a string carry the text of the word or string. The difference between the two is that strings are quoted since they may contain special characters and unquoted words appear in special places in the grammar.

The `KW_` terminals are keywords. They are reserved words that are recognised specially by the lexer. They mark the beginning of major syntactic constructs. Using reserved words helps to eliminate ambiguity in the grammar. The `SYM_` terminals are punctuation symbols.

The `%eop` directive indicates which terminal marks the end of the input. It will be the same as the token returned by the `eof` function in the lexer. It also needs to be repeated in the `%noshift` directive.

The `%arg` directive declares an argument that will be passed into the parser. I use this to pass in the file name for error messages. The `%name` directive provides a prefix for the names of the parser modules. The `%pure` directive declares that all of the action code is side-effect free. If you don't include this then the parser will work harder to compensate which will slow it down. The `%verbose` directive tells ML-Yacc to dump the grammar rules to a `.desc` file. This can be useful for figuring out ambiguity problems but you need to be fairly familiar with LALR parsers.

I've actually combined two grammars together, one for the server configuration file and one for the `.swerve` files. They share a lot of syntax.

Unfortunately ML-Yacc doesn't support more than one start symbol. I have to fake it by prepending a special symbol, called `SYM_SWERVE`, to the terminals from a .swerve file. The parser driver in the Config module will push the string "\001\001\001" onto the front of a `.swerve` file. The lexer will recognise this string as the `SYM_SWERVE` symbol. The parser will then switch to the swerve grammar.

Before looking at the grammar here are the types for the parse tree.

```
structure ConfigTypes =
struct

    datatype Section =
            SectServer of {
                parts:  SectionPart list,
                pos:    Common.SrcPos
                }

        |   SectNode of {
                path:   string,
                parts:  SectionPart list,
                pos:    Common.SrcPos
                }

        (*  This is for the contents of a .swerve file *)
        |   SectSwerve of {
                parts:  SectionPart list
                }

    and     SectionPart = SectionPart of {
                left:   string,
                right:  Literal list,
                pos:    Common.SrcPos
                }

    and     Literal =
                LitIsString of string * Common.SrcPos
            |   LitIsInt of int * Common.SrcPos
end
```

The result of parsing will be a list of sections. Each section contains a list of parts and a part is a "word = value" pair. Every node in the parse tree is annotated with a source position. This gives the file name, line number and

column where the characters corresponding to the node starts. Positions are used in error messages.

Here is the top part of the production section.

```
start:
        section_list                    (section_list)

    |   SYM_SWERVE
        part_list                       ([SectSwerve {
                                            parts = part_list
                                            }])

section_list:
        section                         ([section])

    |   section_list
        section                         (section_list @ [section])


section:
        KW_SERVER
        SYM_LBRACE
        part_list
        SYM_RBRACE                      (SectServer {
                                            parts = part_list,
                                            pos   = KW_SERVERleft
                                            })
    |   KW_NODE
        TOK_WORD
        SYM_LBRACE
        part_list
        SYM_RBRACE                      (SectNode {
                                            path  = TOK_WORD,
                                            parts = part_list,
                                            pos   = KW_NODEleft
                                            })
```

The parsing starts at the first non-terminal which I've called `start`. If it's parsing a server configuration file then the syntax is a list of sections. For a `.swerve` file it is a list of parts.

The action code computes a value for the non-terminal on the left-hand side of a production. A non-terminal on the right-hand side of a production can be

used in the action code and it represents the value of the non-terminal. The notation `KW_SERVERleft` refers to the source position of the first character of the terminal `KW_SERVER`. The notation `TOK_WORD` represents the value carried by the terminal so it's a string variable. For more information on all of this see the ML-Yacc documentation.

Since the `start` non-terminal has the type `(Section list)` its action code must be of this type. So a `.swerve` file will produce a list containing the single SectSwerve section. The production for `section_list` is a standard pattern for a list of one or more things. It's a simple bit of recursion. Note that since ML-Yacc does LALR grammars it must be left-recursion. This means the recursive call to `section_list` appears at the beginning of the second branch of the production. The section production just computes a tree node with type `Section`.

Here's the rest of the grammar. It's quite straight-forward.

```
part_list:
        part                            ([part])

    |   part_list
        part                            (part_list @ [part])


part:
        TOK_WORD SYM_EQUALS
        literal_list
        SYM_SEMICOLON                   (SectionPart {
                                            left  = TOK_WORD,
                                            right = literal_list,
                                            pos   = TOK_WORDleft
                                            })


literal_list:
        literal                         ([literal])

    |   literal_list
        literal                         (literal_list @ [literal])


literal:
        TOK_STRING                      (LitIsString (TOK_STRING, TOK_STRINGleft))
```

```
    |   TOK_INT                         (LitIsInt (TOK_INT, TOK_INTleft))
    |   TOK_WORD                        (LitIsString (TOK_WORD, TOK_WORDleft))
```

The Configuration Lexer

The lexer splits the configuration files up into tokens which are words, strings, symbols and integers. The main difference between words and strings is that that strings can contain any special character so they must be quoted. Words are allowed to contain just enough special characters to form most of the file paths you're likely to want. The symbols include punctuation and some reserved words. The layout of the files is free format with any amount of white space between the tokens.

The lexer is generated using ML-Lex. Starting in the middle of the `config.lex` file are some declarations that are required to interface with the parser.

```
    (*  These definitions are required by the parser.
        The lexer types are supplied by the grammar.
    *)

    type    pos = Common.SrcPos
    type    arg = string                (* type from %arg below *)

    type svalue = Tokens.svalue
    type ('a,'b) token = ('a,'b) Tokens.token
    type lexresult= (svalue,pos) token

    fun eof file = Tokens.EOF(get_pos file 0, get_pos file 0)

%%
%header (functor ConfigLexFun(structure Tokens: Config_TOKENS));
```

ML-Yacc will generate a structure which defines all of the tokens that are passed from the lexer to the parser. These are the terminals of the grammar. The words terminal and token are synonymous. You use `%header` to declare the lexer as a functor that takes the structure as an argument, here called

`Tokens`. Here is the signature for the structure, from the `config.grm.sig` file.

```
signature Config_TOKENS =
sig
type ('a,'b) token
type svalue
val EOF:  'a * 'a -> (svalue,'a) token
val TOK_INT: (int) *  'a * 'a -> (svalue,'a) token
val TOK_STRING: (string) *  'a * 'a -> (svalue,'a) token
val TOK_WORD: (string) *  'a * 'a -> (svalue,'a) token
val SYM_SWERVE:  'a * 'a -> (svalue,'a) token
val SYM_EQUALS:  'a * 'a -> (svalue,'a) token
val SYM_RBRACE:  'a * 'a -> (svalue,'a) token
val SYM_LBRACE:  'a * 'a -> (svalue,'a) token
val SYM_COMMA:  'a * 'a -> (svalue,'a) token
val SYM_SEMICOLON:  'a * 'a -> (svalue,'a) token
val KW_NODE:  'a * 'a -> (svalue,'a) token
val KW_SERVER:  'a * 'a -> (svalue,'a) token
end
```

All of the tokens are defined as functions that map from a pair of source positions and possibly some contained value to the `token` type. There are two source positions so that you can point to the first and last characters of the token in the source file. I just point to the first character and set the second position to be the same as the first. For example to generate the `EOF` token I just call the `Tokens.EOF` function with some dummy source positions.

In the signature the `'a` type variable represents whatever type you choose for the source position. The `svalue` name means "semantic value". It's whatever data will be carried along with the tokens. When used with an ML-Yacc parser it will also include the types for the non-terminals. All you have to do is ensure that there are definitions for the `svalue` and `token` types in the lexer which are equated to the types supplied in the `Tokens` structure. Also you must equate the `lexresult` type to be the same as the parser's token type.

Here is the bottom half of the `config.lex` file which defines the regular expressions for the tokens.

```
%%
%header (functor ConfigLexFun(structure Tokens: Config_TOKENS));
%full
%arg (file: string);

wrd1=[A-Za-z_/\\$:.%+-];
wrd=[A-Za-z0-9_/\\$:.%+-];
word={wrd1}{wrd}*;
str=([^"\n]|\\\n|\\\"|\\\\);
digit=[0-9];
int=[+-]?{digit}+;

ws=[\ \t\013];
%%

"\n"            => (new_line yypos; continue());
{ws}+           => (continue());
#.*\n           => (new_line yypos; continue());


{word}          => (check_reserved yytext file yypos);

{int}           => (fix_integer yytext file yypos);

\"{str}*\"      => (fix_str yytext file yypos);

";"             => (sym Tokens.SYM_SEMICOLON file yypos);
","             => (sym Tokens.SYM_COMMA file yypos);
"{"             => (sym Tokens.SYM_LBRACE file yypos);
"}"             => (sym Tokens.SYM_RBRACE file yypos);
"="             => (sym Tokens.SYM_EQUALS file yypos);
"\001\001\001"  => (sym Tokens.SYM_SWERVE file yypos);


.               => (Log.errorP (get_pos file yypos)
                ["Unrecognised characters in the configuration file."];
                    eof file);
```

The `wrd` definition defines the characters that can appear in a word. The `wrd1` definition is the subset that can be the first character of a word. This excludes digits to avoid confusion with integers. Strings can contain backslash escapes. Within the regular expression a few have to be handled separately. The `\\\n` combination ensures that new-lines are only allowed if they are immediately preceded by a backslash. Similarly a double-quote is

allowed inside a string if it is preceded by a backslash. The last term ensures that the sequence "`foo\\`" is correctly recognised as a backslash at the end of a string and not a backslash followed by an internal double-quote.

The "\001\001\001" is the three CTRL-A character marker that is inserted at the beginning of a `.swerve` file as described in the section called *The Configuration Grammar*.

The lexer uses a few helper functions in the top section of the `config.lex` file to build a token. For example the `yytext` variable contains the complete matched text which for strings will include the double quote characters. The `fix_str` function strips them off and also translates the backslash escapes.

```
fun fix_str yytext file yypos =
let
    val pos = get_pos file yypos
    val chars = explode(substring(yytext, 1, size yytext - 2))

    fun count_nl [] pp = ()
    |   count_nl (#"\n"::rest) pp = (new_line pp; count_nl rest (pp+1))
    |   count_nl (c::rest) pp     = count_nl rest (pp+1)

    fun xlate [] rslt = implode(rev rslt)
    |   xlate (#"\\"::c::rest) rslt =
    let
        val nc =
            case c of
              #"n" => #"\n"
            | #"t" => #"\t"
            | _    => c
    in
        xlate rest (nc::rslt)
    end
    |   xlate (c::rest) rslt = xlate rest (c::rslt)
in
    count_nl chars (yypos+1);
    Tokens.TOK_STRING(xlate chars [], pos, pos)
end
```

According the `Config_TOKENS` signature above the `TOK_STRING` function takes the text of the string as the first argument. The type for this argument comes from the `%term` declaration in the grammar file.

What's a little tricky is keeping track of the line and column positions. I can count lines by being careful to call my `new_line` function (below) for each new-line character in a matched expression. I've made the new-line separate from the white space expression (`ws`) to make it easier to count. ML-Lex generates code to provide the position of a matched regular expression as the character offset from the beginning of the source. This is available in the `yypos` variable. If I save the offset of each new-line then I can work out the column number of a character by subtracting the offset of the character from that of the most recent new-line. This is taken care of in the following code.

```
val     line = ref 1                (* current line *)
val     line_pos = ref 0            (* char position of preceding \n *)

fun get_pos file yypos =
let
    val col = Int.max(yypos - !line_pos, 1) (* see eof *)
in
    Common.SrcPos {file=file, line= (!line), col=col}
end


fun new_line yypos =
(
    line := !line + 1;
    line_pos := yypos
)
```

The `count_nl` function in `fix_str` above is needed to account for new-line characters embedded in strings. It has to track the source position within the string to keep the positions right.

Integers as strings are converted to numeric values in the `fix_integer` function. The `sym` function just adds source positions to the symbols. I won't show these here as they are simple enough. Reserved words are filtered out in the `check_reserved` function.

```
val reserved_words = [
    ("SERVER",      Tokens.KW_SERVER),
    ("NODE",        Tokens.KW_NODE)
    ]
```

```
fun check_reserved yytext file yypos =
let
    val uword = Common.upperCase yytext
    val pos = get_pos file yypos
in
    case List.find (fn (w, _) => w = uword) reserved_words of
      NONE          => Tokens.TOK_WORD(yytext, pos, pos)
    | SOME (_, tok) => tok(pos, pos)
end
```

Since there are only a few a search through a list is fine. The word matching is case-insensitive.

The Parser Driver

This section completes the description of the parser by showing how it is used in the Config module. The various structures and functors are assembled to make a complete parser as follows.

```
(*  Assemble the pieces to make a parser. *)

structure ConfigLrVals = ConfigLrValsFun(structure Token = LrParser.Token)
structure ConfigLex    = ConfigLexFun(structure Tokens = ConfigLrVals.Tokens)
structure ConfigParser = JoinWithArg(structure LrParser = LrParser
                        structure ParserData = ConfigLrVals.ParserData
                        structure Lex = ConfigLex)

(*  Max number of tokens to lookahead when correcting errors. *)
val max_lookahead = 15;

(*  The syntax error messages use the token names. This is for editing
    them to something more readable.
*)
val syntax_edits = [
    ("KW_SERVER",       "Server"),
    ("KW_NODE",         "Node"),
    ("SYM_SEMICOLON",   "semicolon"),
    ("SYM_COMMA",       "comma"),
    ("SYM_LBRACE",      "'{'"),
    ("SYM_RBRACE",      "'}'"),
    ("SYM_EQUALS",      "'='"),
```

```
    ("TOK_WORD",        "word"),
    ("TOK_STRING",      "string"),
    ("TOK_INT",         "number")
    ]
```

The `ConfigLrVals` structure contains the tables of parsing operations for the grammar. The `ConfigLex` structure contains the complete lexer specialised with the types needed to communicate with the parser. The `JoinWithArg` functor is part of the ML-Yacc library. It joins all the pieces together. The "WithArg" part of the name indicates that it supports an argument being passed in at parsing time. I use this to carry the file name. You can see it in the `%arg` declarations in the grammar and lexer files. (The file argument isn't used in the parser but the way the joining works it must be there if the lexer has one).

The result is a complete parser in the `ConfigParser` structure which will be used below.

A parser generated by ML-Yacc will do syntax correction. This means that if there is a syntax error it will attempt to insert or delete tokens to change the input into something parsable and then continue. It will produce an error message showing what change it made which should give the user an idea of what the original syntax error was. This sounds clever but it can be confusing. It means that if you omit a semicolon for example you will get an error message saying that one was inserted. This isn't very user-oriented. What's worse is that the tokens in the messages are described using the names that appear in the grammar file. I like to have distinctive terminal names in the grammar file, in uppercase. So I process the error messages to convert the terminal names to something more readable. The `syntax_edits` list in the above code is used for this processing. The `max_lookahead` parameter controls the error correction. The value 15 is recommended in the ML-Yacc documentation for most purposes.

Here is the `parse_config` function which drives the parser.

```
and parse_config swerve file: Section list =
let
    fun parse_error(msg, pos1, pos2) = Log.errorP pos1 [edit_errors msg]
```

```
    val swerve_done = ref false

    fun input rstrm n =
    (
        if swerve andalso not (!swerve_done)
        then
            (swerve_done := true; "\^A\^A\^A")
        else
            TextIO.inputN(rstrm, n)
    )

    fun do_parser holder =
    let
        val rstrm = TextIOReader.get holder

        fun do_parse lexstream =
            ConfigParser.parse(max_lookahead, lexstream, parse_error, file)

        val in_stream = ConfigParser.makeLexer (input rstrm) file

        val (result, new_stream) = do_parse in_stream
                    handle ParseError => ([], in_stream)
    in
        TextIOReader.closeIt holder;
        result
    end
in
    case TextIOReader.openIt' file of
      NONE   => []
    | SOME h => do_parser h
end
```

The `TextIOReader.openIt'` function opens the file without worrying about connection time-outs. The `swerve` argument indicates that a `.swerve` file is being parsed. The `.swerve` files are actually read while the server is processing connections so I should be worrying about time-outs for them but the files are small so I'll get away with it.

In the `do_parser` function the parsing is run by a call to the ConfigParser.parse function. The second argument to `ConfigParser.parse` is the lexer. The third is a call-back function for error messages and the fourth is the `%arg` argument value. The lexer is made with

the `ConfigParser.makeLexer` function which takes a source reading function and a `%arg` argument value. The `input` function delivers the contents of the file in chunks of size `n`. If it is a `.swerve` file then the first chunk is forced to be the triple CTRL-A marker.

I've omitted a description of the syntax error editing. It's just some straight-forward string manipulation. The `Substring.position` function does the job of finding the string to replace.

The end-result of all of this is parse tree whose type is `ConfigTypes.Section list`, which is the type of the `start` non-terminal in the grammar.

Processing the Parse Tree

The output from the parser is a list of sections of the type `ConfigTypes.Section`. The two main processing steps that follow are for the server and node sections. Refer to the `processConfig` function in the section called *The Config Module - Interface*.

The `process_server_section` function looks through the sections for the server configuration section.

```
and process_server_section sections =
let
    fun match (SectServer _) = true
    |   match _              = false

    val sects = List.filter match sections
in
    case sects of
      [SectServer {parts, ...}] => process_server_parts parts

    | [] => (Log.error
               ["A server configuration section must be supplied."];
             raise Bad)

    | _  => (Log.error
               ["There are multiple server configuration sections."];
             raise Bad)
```

```
end
```

The `process_server_parts` function saves each parameter into the static variables. (This design neatly allows the server section to be anywhere in the file).

The `process_node_sections` function finds each node section and adds it to the list of node configurations in a static variable. The static variables are described in the section called *The Config Module - Interface*.

```
and process_node_sections sections =
let
    fun process sect =
    (
        case process_node_section sect of
          NONE        => ()
        | SOME config => cf_nodes := config :: (!cf_nodes)
    )

    fun match (SectNode _) = true
    |   match _            = false

    val sects = List.filter match sections
in
    case sects of
      [] => (Log.log Log.Warn (TF.S
                "There are no node configuration sections."))

    | _  => app process sects
end
```

I won't describe the parameter processing in detail. It's a lot of long-winded checking of values for the correct format and legality. I'll just describe the general idea.

The `process_server_parts` takes a list of `ConfigTypes.SectionPart` values which contain one parameter each. It has two dispatch tables for string-valued and integer-valued parts. These dispatch to functions that check the value and return the value if it is legal. If the value is illegal then an error message is logged and the `Bad` exception is raised. This is caught at

the top in `processConfig` which aborts the whole server with a `FatalX` exception.

Each dispatch step reduces the list of parts by removing those that are recognised. If there are any parts left over then they must be unrecognised parameters and error messages are logged. (See the `unrec_param` function).

The server configuration record is built up from the successful values returned from the dispatch functions. Utility functions are used to extract particular parameters. For example the `reqstr` function finds a required parameter that has a string value. Defaulting is performed at this stage.

Node section processing is similar but since there are fewer parameters and their types are more complex there isn't a formal dispatch table. There isn't as much checking for legality as there should be. For example I don't check that the name of a built-in handler is legal at this stage, nor if a CGI script exists.

MIME Type Configuration

The server configuration contains the path to the MIME types file. This has the same format as is used by Apache. The contents of this file is read in by the `process_mime_file` function and saved into a table. The table maps from a file extension to a pair of major/minor MIME type names. For case-insensitivity the extensions are saved in upper case.

```
(*  The mime information is just a map from an extension to
    the pair.
*)
val mime_table: (string * string) STRT.hash_table =
                                    STRT.mkTable(101, NotFound)
```

There is only one external API function, `lookupMime`.

```
and process_mime_file() =
let
    val ServerConfig {mime_file, ...} = getServerConfig()
in
```

```
    if Files.readableReg mime_file
    then
        FileIO.withTextIn (Abort.never())
            mime_file () (read_mime mime_file)
    else
        Log.error ["The MIME types file is not readable: ", mime_file]
end



and read_mime mime_file stream : unit =
let
    fun loop lnum =
    (
        case TextIO.inputLine stream of
          "" => ()
        | line => (do_line lnum line; loop (lnum+1))
    )
... omitted material ...
in
    loop 1
end


and lookupMime ext = STRT.find mime_table (upperCase ext)
```

The Common Layer

This layer collects various utility functions such as file I/O and low-level systems such as logging. The largest modules here deal with resource allocation (see the section called *System Resource Management* in Chapter 8). These are the Open File Manager and the Temporary File Manager. I'll describe the modules in alphabetical order.

The Abort Module

This module implements an `Abort` type that signals that connection processing should be aborted. This will be triggered by a time-out. An abort can also be forced before the time-out happens which is useful if a connection is found to be broken while trying to do I/O.

There are two ways the abort condition can be detected. It can be polled or a CML event can be obtained for use in a `CML.select` call. Here is the module API.

```
signature ABORT =
sig
    type Abort

    (*  The arg is the time-out in seconds. *)
    val create:     int -> Abort

    (*  This never times out. *)
    val never:      unit -> Abort

    (*  This returns an abort event for synchronising. *)
    val evt:        Abort -> unit CML.event

    (*  This tests if the event has occurred. *)
    val aborted:    Abort -> bool

    (*  This forces the abort to happen early even if it is the 'never'
        condition.
    *)
    val force:      Abort -> unit
end
```

In an earlier design I implemented the time-out by starting a new thread for each connection to wait on a time-out event. The thread was used to set the flag that was tested by the `aborted` function. But since these timer threads lasted for as long as the time-out period there would be large numbers of them hanging around in the server during a heavy load of hundreds of connections per second. The performance of the CML run-time does not scale very well for large numbers of time-out events, as I explained in the section

called *Behaviour of Timeout Events* in Chapter 7. After a few seconds of heavy load I found the server grinding to a halt.

The current implementation uses an I-variable from the synchronous variables module (see the section called *Synchronous Variables* in Chapter 6. The I-variable has all of the necessary properties for an abort condition. It can be set and tested like a flag and an event is available to indicate when it is set. Here is the definition for the `Abort` type.

```
datatype Abort = Abort of unit SyncVar.ivar
```

To manage the timing I've included a manager thread that maintains a map from future points in time to lists of `Abort` values.

```
structure Map = IntRedBlackMap

datatype Request =
        Add of int * Abort          (* (timeout, force) -> abort *)

datatype State = State of {
    time:       int,                (* seconds since startup *)
    live:       (Abort list) Map.map(* waiting to expire *)
    }


fun server ch () =
let
    val start = Time.now()

    fun toTime secs = Time.fromSeconds(LargeInt.fromInt secs)
    fun trunc  time = Int.fromLarge(Time.toSeconds time)


    fun loop (state as State {time, ...}) =
    let
        fun request (Add (delay, abort)) = add delay abort state


        (*  If the timing drifts off it won't hurt if this
            event is for a time in the past. It will be immediately
            enabled.
        *)
        val time_evt = CML.atTimeEvt(Time.+(start, toTime(time+1)))
```

```
        val new_state = CML.select[
                CML.wrap(CML.recvEvt ch,
                    MyProfile.timeIt "abort request" request),

                CML.wrap(time_evt,
                    (*MyProfile.timeIt "abort expire"*) (expire state))
                ]
    in
        loop new_state
    end
... omitted material ...
in
    loop (State {time = 0, live = Map.empty})
end



structure Mgr = Singleton(
                        type input    = Request CML.chan
                        val  newInput = CML.channel
                        val  object   = server
                        )
```

The Add request inserts an abort into the manager's map.

```
and add delay abort (state as State {time, live}) =
let
    (*  Find out the end-time in seconds relative to
        the start time of the server, rounded to the
        nearest second.
    *)
    val now   = Time.now()
    val since = Time.-(now, start)
    val ends  = trunc(Time.+(
                    Time.+(since, toTime delay),
                    Time.fromMilliseconds 500
                    ))

    val _ = Log.testInform Globals.TestTimeout Log.Debug
            (fn()=>TF.L ["Abort add delay=", Int.toString delay,
                         " now= ", Time.fmt 6 now,
                         " ends=", Int.toString ends
                         ])

    (* The insert operation will either insert or replace. *)
    fun add_abort() =
```

```
    (
        case Map.find(live, ends) of
          NONE =>
            let
                val new_live = Map.insert(live, ends, [abort])
            in
                State {time=time, live=new_live}
            end

        | SOME ab_list =>
            let
                val new_live = Map.insert(live, ends, abort::ab_list)
            in
                State {time=time, live=new_live}
            end
    )
in
    add_abort()
end
```

The key for the map is the the expiry time for the time-out measured in seconds since the start of the manager. The `add_abort` function either creates a new entry or adds the `Abort` to an existing entry. The maximum number of entries will be the size of the time-out from the server's configuration plus 1. This number should stay reasonably small. A 60 second time-out would be reasonable.

The manager counts through the seconds since startup. At each second the `expire` function scans the keys of the map to see if any lists have expired.

```
and expire (state as State {time, live}) () =
let
    (*  Find out what the time really is. *)
    val count = trunc(Time.-(Time.now(), start))

    fun check_entry (at_time, ab_list, new_live) =
    (
        if count >= at_time
        then
        (
            Log.testInform Globals.TestTimeout Log.Debug
                (fn()=>TF.L ["Abort expiring, count=",
                             Int.toString count,
                             " live size=",
```

```
                            Int.toString(Map.numItems live)
                            ]);

            (* Remove the entry and set all its aborts. *)
            app set_ab ab_list;
            new_live
        )
        else
            (*  Put the entry back into the map. *)
            Map.insert(new_live, at_time, ab_list)
    )


    and set_ab (Abort ivar) = (SyncVar.iPut(ivar, ()))
                                    handle _ => ()

    val new_live = Map.foldli check_entry Map.empty live
in
    State {time=count, live=new_live}
end
```

Since the red-black map is a pure functional value the `expire` function has to build a new map at each scan. This won't be a burden since it only happens once a second and the number of entries is not large. During the building the expired lists are simply omitted. The I-variables in the lists are set and then the `Abort` values are released. If there is still a client connection with a reference to an expired `Abort` it can poll it or wait on its event. Other `Aborts` simply become garbage.

Here is the implementation of the API functions.

```
fun create delay =
let
    fun run() =
    let
        val abort = Abort (SyncVar.iVar())
    in
        CML.send(Mgr.get(), Add(delay, abort));
        abort
    end
in
    MyProfile.timeIt "abort create" run ()
end
```

```
fun evt     (Abort ivar) = SyncVar.iGetEvt ivar
fun aborted (Abort ivar) = isSome(SyncVar.iGetPoll ivar)

fun force   (Abort ivar) = (SyncVar.iPut(ivar, ()))
                            handle _ => ()

fun never() = Abort (SyncVar.iVar())
```

The `force` function just sets the I-variable directly. If it is already set then this is ignored.

The `never` value is useful for times when you know there won't be a time-out, for example during the startup of the server. Since it can be forced, every caller must get a distinct value.

The Common Module

This gathers miscellaneous small declarations that are useful through-out the server. The module is normally opened where-ever it is used to avoid qualifying its declarations with a `Common.` prefix. So I don't want too many declarations which increases the risk of clashing with other identifiers in the server.

The following declarations are involved in aborting the server.

```
exception FatalX
exception InternalError of string

fun toErr s = (TextIO.output(TextIO.stdErr, s);
               TextIO.flushOut(TextIO.stdErr))

(*  These shutdown the server with the given status code. *)
fun success() = RunCML.shutdown OS.Process.success
fun fail()    = (toErr "Aborting\n"; RunCML.shutdown OS.Process.failure)
```

The `FatalX` exception aborts processing in the main thread when some . The `InternalError` exception is for filling in impossible cases in the code (fingers crossed). The `success` and `fail` functions will shutdown the server

returning an appropriate process status. In either case the `Startup.finish` code is run (see the section called *The Startup Module*).

This module has the declaration of the `SrcPos` type for describing the location of an error in a configuration file. There is also `STRT`, the common hash table over string keys.

The FileIO Module

This module contains utility functions that operate on disk files and directories. They are typically wrappers around the SML Posix functions which log errors for the server. The operations include removing and creating a file, getting some file properties such as the size and modification time and doing controlled reading of files and directories.

The last of these are non-trivial. The `withTextIn` function controls the reading of a text file using the Open File Manager.

```
fun withTextIn abort file default func =
let
in
    case TextIOReader.openIt abort file of
      NONE   => default               (* the open failed *)
    | SOME h =>
        ((func (TextIOReader.get h))  (* handle an I/O failure with closing *)
            handle x => (Log.logExn x; TextIOReader.closeIt h; default)
        ) before (TextIOReader.closeIt h)
end
handle x => (Log.logExnArg file x; default)
```

The calling thread will be blocked until a file descriptor is available. The abort of a connection due to a time-out is detected. Care is taken to ensure that neither I/O errors nor an abort leave the file open. The caller supplies a function `func` that reads the file via a `TextIO.instream`. The result from this function will be returned if all goes well. If there is an error then the `default` value will be returned. The SML `General.before` function is useful for attaching a clean-up operation like closing a file to an expression.

The `listDir` function does a similar job but returns a list of the files in a directory, excluding the dot and dot-dot entries. An empty list is returned if there is an error.

```
fun listDir abort dir =
let
    fun loop strm rslt =
    (
        case OS.FileSys.readDir strm of
          "" => rslt
        | s  => loop strm (s::rslt)
    )
in
    case DirReader.openIt abort dir of
      NONE => []                (* the open failed *)

    | SOME h =>
        (loop (DirReader.get h) []) before (DirReader.closeIt h)
end
handle x => (Log.logExnArg dir x; raise x)
```

The `exclCreate` function for creating lock files is in this module. There is a description of it in the section called *The Startup Module*.

The Files Module

This module contains simple utilities that manipulate file names and test properties of files. For example there are the `dirName`, `baseName`, `splitExt` and `appendFile` functions to break up and build file paths. These are simple wrappers for the SML OS.Path functions. They don't need any further description.

The property testing functions are wrappers around the SML Posix.FileSys functions that do a similar job to the Unix `stat` and `access` system calls. Here are some of the functions. `FS` is an abbreviation of Posix.FileSys.

```
fun exists path = FS.access(path, [])

fun isDir  path = exists path andalso FS.ST.isDir(FS.stat path)
fun isReg  path = exists path andalso FS.ST.isReg(FS.stat path)
```

```
fun isSym  path = exists path andalso FS.ST.isLink(FS.stat path)

fun accessibleDir path =
(
    isDir path andalso FS.access(path, [FS.A_READ, FS.A_EXEC])
)
```

The Log Module

This implements the Logging Manager. It writes time-stamped messages to either standard error or a log file. The messages can have different severity levels in the usual way. The severity threshold level can be set with the `LogLevel` server configuration parameter (see the section called *The Server Parameters* in Chapter 8).

The manager is needed to ensure that messages from different threads in the server are logged atomically, that they don't get their fragments interleaved. I also want the server threads to not be held up while logging messages. A thread should be able to send off a message and then immediately continue with its work. The manager implements this by using a CML mailbox to receive messages. A mailbox has unlimited buffering so no send operation will ever block. This might introduce a denial-of-service risk if a client connection can be induced to generate large numbers of errors rapidly. But I think that the risk is miniscule.

Here is the basic logging API. Messages are composed from text fragments (see the section called *The Text Module*) so that they can be built efficiently.

```
type Level

val Fatal: Level
val Error: Level
val Warn:  Level
val Info:  Level
val Debug: Level

(*  This writes a logging message.
    Error messages are counted.
```

```
*)
val log:        Level -> TextFrag.Text -> unit

(*  This writes a logging message related to some source file position.
    Error messages are counted.
*)
val logP:       Level -> Common.SrcPos -> TextFrag.Text -> unit
```

The log level argument is tested and the message is discarded if the severity is below the threshold. This can be wasteful if the message will usually not be logged, such as for informational or debugging messages. For these I have a slightly different API.

```
val inform:     Level -> (unit -> TextFrag.Text) -> unit
val testInform: int -> Level -> (unit -> TextFrag.Text) -> unit
```

Here the message is represented by a function that generates the text. The function won't be called if the message is not logged so the cost of generating the message is avoided. A typical use of this is the debugging message:

```
val _ = Log.inform Log.Debug (fn()=>TF.L
            ["HTTP reading into file len=", Int.toString len])
```

The integer to string conversion and assembly of the message will not be done unless debugging messages are being logged.

The logging destination and level are controlled by these functions.

```
(*  This returns the count of error messages seen so far.
*)
val numErrors:  unit -> int

(*  This waits until the log messages have drained.
*)
val flush:      unit -> unit

(*  Set the file for error logging.  This is currently only
    allowed to be set once.
*)
val setLogFile: string -> unit

(*  Set the level for error logging. *)
val setLevel:   Level -> unit
```

```
(*  Set the level for error logging to be at least as low as the
    given level.
*)
val lowerLevel:   Level -> unit
```

All messages with severity of `Error` or greater are counted. The server startup code calls the `Log.numErrors` function to see if errors were reported while reading the configuration files. If so then it aborts the server with a fatal error.

The `lowerLevel` function is used while reading the server configuration files to lower the log level to `Warn` to ensure that warning messages can be seen. The `flush` function is needed when changing the logging destination or level.

Another important source of logging messages are exceptions. Here is the API for logging exceptions.

```
(*  Log any kind of exception.
    This is guaranteed not to raise any exception.
*)
val logExn:  exn -> unit

(*  Log with some extra information e.g. a file name. *)
val logExnArg:  string -> exn -> unit
```

The `logExn` function itself must not raise any exceptions otherwise code such as the following from the Listener module will go wrong when the handler fails to complete.

```
handle x =>
    let
        (*  See also Connect.getPort *)
        val (_, port) = INetSock.fromAddr sock_addr
    in
        (
            Socket.close sock;
            TmpFile.releasePort port
        ) handle _ => ();               (* being paranoid *)
        Log.logExn x;
        CML.send(lchan, ConnDied)
```

```
    end
```

The manager is a singleton object (see the section called *The Singleton Module*) running a server thread for an internal protocol. Here is the top-level of this server.

```
datatype LogRequest =
        ReqLog of Level * TF.Text * Time.time
    |   ReqSetFile of string
    |   ReqNumErrors of int Sy.ivar
    |   ReqFlush of unit Sy.ivar        (* flush with acknowledge *)



fun log_server mbox () =
let
    (*  An imperative state will be OK in this small context. *)
    val num_errors: int ref = ref 0
    val log_strm  = ref TextIO.stdErr
    val log_file  = ref ""


    fun loop() =
    let
        fun timeout() = TextIO.flushOut(!log_strm)
    in
        CML.select[
            CML.wrap(Mailbox.recvEvt mbox, handle_msg),
            CML.wrap(CML.timeOutEvt (Time.fromSeconds 1), timeout)
            ];
        loop()
    end
```

The manager's state consists of the number of errors, and the output stream for logging. (The log file name is not used in the code but might be useful later). I've been a bit lazy and implemented the state using imperative variables. This saves winding the state through all of the code especially as it is rarely updated. Since only the server thread updates them they are safe.

I've included a 1 second time-out in the server. This flushes the log stream to disk so that error messages show up promptly.

Here is the protocol handler.

```
and handle_msg msg =
(
    case msg of
      ReqLog (level, msg, time) => internal_log level msg time

    | ReqSetFile file =>
        (
            if !log_file = ""
            then
                set_log_file file
            else
                internal_log Error (TF.S
                    "Attempted to set the log file twice")
                    (Time.now())
        )

    | ReqNumErrors rvar => Sy.iPut(rvar, !num_errors)

    | ReqFlush rvar => Sy.iPut(rvar, ())
)
```

The `ReqNumErrors` message is a request to return the number of errors. The `ReqFlush` handshakes with the `flush` function below to make sure that the mailbox is empty.

```
structure Logger = Singleton(
                        type input    = LogRequest Mailbox.mbox
                        val  newInput = Mailbox.mailbox
                        val  object   = log_server
                        )

fun send_request req = Mailbox.send (Logger.get(), req)

fun flush() =
let
    val rvar = Sy.iVar()
in
    send_request (ReqFlush rvar);
    Sy.iGet rvar
end
```

The `CML.recv` call in `flush` will block until the server thread responds to the `ReqFlush` message. All preceding messages in the mailbox must have been processed at this point.

Here is the function that actually prints the message.

```
and internal_log level msg when =
let
    fun put s = TextIO.output(!log_strm, s)

    val date  = Date.fromTimeLocal(when)
    val fdate = Date.fmt "%Y %b %d %H:%M:%S" date
in
    put(concat[fdate, " ", formatLevel level, ": "]);
    TF.appPrefix "\t" put msg;
    put "\n";
    update_counts level
end
```

I add a simple time-stamp to each message. The `appPrefix` call applies the `put` function to each fragment of the message but putting a tab before each subsequent line. This lays out multi-line messages nicely as long as the TextFrag line-breaking is used properly.

Here is the implementation of the basic `log` API.

```
fun log level msg =
(
    if level >= (!log_level) orelse level = Fatal (* speed up check *)
    then
        send_request (ReqLog (level, msg, Time.now()))
    else
        ();

    if level = Fatal
    then
    (
        flush();
        Common.toErr(concat[formatLevel level, ": ",
                            TF.toString TF.UseLf msg, "\n"]);
        Common.fail()               (* abandon execution *)
    )
    else
        ()
)
```

There is a potential race condition in that the `log_level` variable could be set by more than one thread calling `setLevel` at the same time. But in

practice the level is only set at configuration time when only one thread is running.

Here is the implementation of the exception logging API.

```
fun logSysErr arg (msg: string, _: OS.syserror option) = error [arg, " ", msg]

fun logExn x = logExnArg "" x

and logExnArg arg x =
(
    case x of
      OS.SysErr x        => logSysErr arg x
    | IO.Io {cause, ...} => logExnArg arg cause

    | InternalError msg  => log Fatal (TF.L ["Internal Error: ", msg])
    | FatalX             => log Fatal (TF.S "Fatal Error")

    | x                  => log_any x
)
handle _ => ()              (* don't raise any more exceptions *)

and log_any x = log Error (TF.L [exnName x, ": ", exnMessage x])
```

This formats all kinds of exceptions. The system and I/O error exceptions are the most likely and they come with extra detail. As a fall-back I can always report the name of the exception with `General.exnName`.

The Mutex Module

The Singleton pattern (see the section called *The Singleton Module*) relies on having a static variable that holds a handle to the singleton object. This is updated with the handle the first time that it is accessed. Since it can be accessed by any number of threads I have the classic race-condition problem. I need some sort of mutual exclusion (mutex) to control access to these static variables. The Mutex module implements mutexes using an M-variable as described in the section called *Semaphores via Synchronous Variables* in Chapter 6. Here is the API for the module.

```
signature MUTEX =
```

```
sig
    type Mutex

    (*  Mutex values can be saved over an exportML so you can
        "statically" create them.
    *)
    val create:     unit -> Mutex

    (*  This runs the function in the critical section.
        This will work with CML running or not.
    *)
    val lock:       Mutex -> (unit -> 'a) -> 'a
end
```

A mutex can be created as a top-level value in a module. This can be saved in an exported heap without any trouble. The `lock` function is passed a job function that typically updates some static variable. Here is the implementation.

```
structure Mutex: MUTEX =
struct
    structure SV = SyncVar

    type Mutex = bool SV.mvar


    fun create() = SV.mVarInit true


    fun lock mutex func =
    (
        SV.mTake mutex;
        let
            val r = func()
        in
            SV.mPut(mutex, true);
            r
        end
        handle x => (
            SV.mPut(mutex, true);
            raise x
            )
    )

end
```

The MyProfile Module

This module implements two utilities. The first is a simple run-time timer to measure how many microseconds it takes to run a function. I use this to get some idea of how long the server spends performing each step in returning a page. On my Linux ix86 machine the timer has a resolution of 1 microsecond. It can time functions down to around 5 microseconds with reasonably reliable results.

The second utility is some code for profiling. The standard profiling code described in the section called *Execution Time Profiling* in Chapter 4 will not link with CML since it uses code within the SML/NJ compiler that is linked with the non-CML TextIO module. I have repeated the profiling report function here with some simplifications. It can produce a profiling report on `stdout` when the server has been compiled with profiling.

The Open File Manager

This module implements the resource management for file descriptors that is described in the section called *System Resource Management* in Chapter 8. If a connection cannot get enough file descriptors then I could either abort the connection or make it wait until more are available. Aborting is a bit crude. With the concurrent design I should be able to get the parts of the server to cooperate better than that.

The Open Manager is a central place where the usage of file descriptors is counted. Before a connection attempts to open files it must request the Open Manager to allocate it the number of descriptors that it will need. If there are not enough free then the connection must wait. The connection will be put onto a queue. When another connection closes its files the descriptors will be returned to the Open Manager. The Manager will pass them onto a waiting connection. The waiting connections are dealt with in first-in-first-out order for fairness.

If a connection is aborted or fails with an internal exception then there is a risk that files will be left open and the server will "leak" descriptors and eventually grind to a halt. I already rely on the SML garbage collector to clean up a connection if there is a time-out or another abort condition. I want the open files to be cleaned up as well. This is an application of finalisation as described in the section called *Weak Pointers* in Chapter 4.

I don't want to rely on finalisation to close files during normal operation. This would leave files open unnecessarily until the next garbage collection. The Open Manager must allow files to be opened and closed normally but also detect when an open file has become garbage and close the file and make the file descriptor available for reuse.

A socket for an incoming connection is a special case. Its file descriptor is created by the operating system. The best that the server can do is make sure that it is counted by the Open Manager after the connection is established.

Being Generic

The Open Manager must be able to deal with all of the different kinds of objects that require file descriptors. These include sockets, regular files (binary and text), directories and pipes to communicate with child processes. Each of these kinds has a different way of opening and closing.

I want the Manager to be extensible. It should be easy to add new kinds of file objects. This is something that object-oriented languages do well. In an O-O language I would define an abstract base class for a file object and sub-class it for each kind. The language would dynamically dispatch an `open()` method on a file object depending on the actual kind of the resource.

Unfortunately SML does not provide any form of dynamic dispatch. I could revert to a variant-record structure with a datatype like

```
datatype Object =
        Regular of ...
    |   Directory of ...
    ...
```

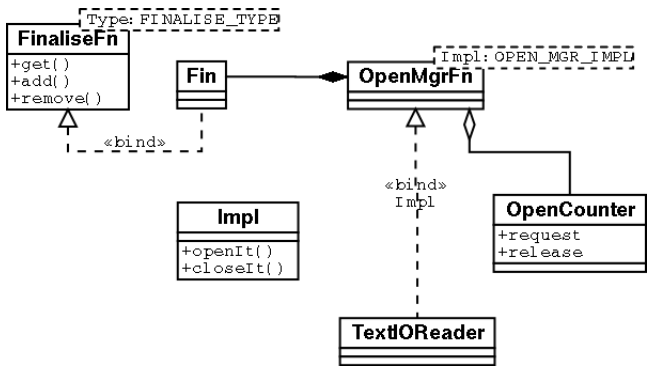
But then I would have to have `case` expressions all over the place where the file object is handled. This is very messy and hard to extend. This is the very problem that the object-oriented paradigm is designed to solve.

An alternative structure is to divide the Manager along the boundaries between kinds. I will create separate managers specialised to each kind of file object. These managers must cooperate to keep a count of the open files so there is yet another Open Counter manager that just counts the files. It does not need to know anything about operating on the files.

The generic manager, `OpenMgrFn`, is a functor specialised to its kind of file with an implementation module. The resulting kind-specific manager needs its own specialised finalisation manager, `Fin`, since the finalisation needs to know how to close any open files that it finds.

The resulting class diagram is shown in Figure 9-2.

Figure 9-2. The Open Manager Objects.



The generic functor is organised like this

```
functor OpenMgrFn(
    structure Impl: OPEN_MGR_IMPL
    ) : OPEN_MGR =
struct
    open Common
```

```
    structure TF = TextFrag

    structure Ctr  = OpenCounter
    structure Impl = Impl

    structure Fin = FinaliseFn(
                        structure Type =
                        struct
                            type T = Impl.Opened * Ctr.Allocation
                            fun finalise (opn, _) = ignore(Impl.closeIt opn)
                            val name = Impl.name
                        end)


    type Arg    = Impl.Arg
    type Opened = Impl.Opened
    type Closed = Impl.Closed
    type Holder = Fin.Holder
```

It takes an implementation structure parameter called `Impl`. The functor builds a specialised finaliser that it calls `Fin`.

An implementation conforms to the following signature.

```
signature OPEN_MGR_IMPL =
sig
    val     name:   string

    type    Arg
    type    Opened
    type    Closed

    (*  This is the number of file descriptors that are needed
        by the open.
    *)
    val num_fds:    int


    datatype Result =
            Success of Opened
        |   Fail                (* give up totally *)
        |   Retry               (* should try again later *)

    val openIt:     Arg -> Result
    val closeIt:    Opened -> Closed
```

```
end
```

The `openIt` function must open the object and (if successful) return a type `Opened` which represents the opened object. The `Result` type is used by the handshaking protocol with the Open Counter manager which is described in the section called *Opening a File*.

Finally the Open Manager produces this signature.

```
signature OPEN_MGR =
sig
    structure Impl: OPEN_MGR_IMPL

    (*  This describes what can be opened or closed. *)
    type    Arg = Impl.Arg

    (*  This represents an open object. *)
    type    Opened = Impl.Opened

    (*  This is the type returned from a close operation. *)
    type    Closed = Impl.Closed

    (*  This is a holder for the object.  The object will be
        finalised if the caller loses its reference to the
        object.
    *)
    type    Holder

    val get:    Holder -> Opened

    (*  Open/close the object.
        This will return NONE if the open failed or was aborted.
    *)
    val openIt:     Abort.Abort -> Arg -> Holder option
    val openIt':    Arg -> Holder option
    val closeIt:    Holder -> Closed
end
```

The design of these managers depends on the signature constraints being transparent. Transparent means that information about the implementation of a type is known by the compiler and allowed to propagate through the various modules. The `ExecReader` module relies on this. If you follow through the declarations you find that the following types are identical.

```
ExecReader.Opened = ExecReader.Impl.Opened = Unix.proc * string
```

The code for the CGI Node Handler in the section called *The CGI Node Handler* can extract the `Unix.proc` value to manipulate the process by writing

```
val (proc, _) = ExecReader.get holder
```

The opposite of transparent is an opaque signature constraint which is indicate by using "`:>`" instead of "`:`" before the signature name. With an opaque the implementation of the `Opened` type would be hidden since only the name is declared in the `OPEN_MGR_IMPL` signature.

Finalisation

Finalisation is done by maintaining a collection of weak references to each open file. To make this work I have to have one value that is shared between the finalisation manager and a client. The manager keeps a weak reference to this value and the client has one or more normal (strong) references. When all of the strong references have gone the value will be collected and then the weak reference will report that the value is gone. This will be a trigger for the manager to close the file.

Note that the shared value is *not* the open file. The manager must still have a reference to the file after the shared value has been collected. The client must be careful not to hold a reference to the open file without also having one to the shared value. To make this safer the client will only be able to get to the open file from the shared value.

A finalisation manager has the following signature. The `Holder` type will be chosen to ensure that it is always copied by reference.

```
signature FINALISER =
sig
    (*  This is the value that is shared between the client
        and the manager.
    *)
```

```
    type Holder

    (*  This is the value in the holder that will be finalised.  *)
    type T

    val get:    Holder -> T

    (*  This adds a new T to the manager.  *)
    val add:    T -> Holder

    val remove: Holder -> unit
end
```

The generic code for the manager is in a functor which takes the details about the type of the finalised value as a parameter.

```
signature FINALISE_TYPE =
sig
    type T

    val finalise:   T -> unit
    val name:       string
end

functor FinaliseFn(
    structure Type: FINALISE_TYPE
    ): FINALISER =
struct
```

A manager is a concurrent object with a simple list of weak references as its state. It takes messages to add and remove values from its list. Its message protocol is

```
datatype Req =
        ReqAdd of T * Holder Sy.ivar
    |   ReqRemove of Holder

(*  When the holder is collected we should have the last
    strong ref to T which we finalise.
*)
type Wref  = int * T * (Holder W.weak)

(*  This requires a linear scan of all held objects which
    shouldn't be a performance problem since GCs are
```

```
    infrequent.
*)
type State = int * Wref list
```

The add message creates the holder. The holder contains a pair of an integer key and the value to be finalised. The integer key allows holders to be identified since we can't assume that the value supports the equality operator. The key is also applied to the weak references.

```
type    T = Type.T

type Pair = int * T

(*  We use a ref on the Pairs to ensure that they are
    copied by reference.
*)
type Holder = Pair ref
```

The manager must receive a signal telling it when the garbage collection has been done. This is received as a message from the signal manager (see the section called *The Signal Manager*).

```
fun server chan () =
let
    val gc_port = SignalMgr.mkGcPort()

    fun loop state =
    (
        loop(CML.select[
            CML.wrap(CML.recvEvt chan, handle_msg state),
            CML.wrap(SignalMgr.gcEvt gc_port, finalise state)
            ])
    )
in
    loop(0, [])
end
```

The `finalise` function scans the weak references and tests which ones to keep.

```
and finalise (state as (tag_cnt, wrefs)) () : State =
let
    val _ = Log.testInform G.TestFinalise Log.Debug
```

```
                (fn()=>TF.L ["Finaliser ", Type.name, ": finalising"])

    (*  Test if this wref should be kept or finalised.
    *)
    fun keep (_, value, wref) =
    (
        case W.strong wref of
          NONE   => (Type.finalise value; false)
        | SOME _ => true
    )
in
    (tag_cnt, List.filter keep wrefs)
end
```

Opening a File

Opening a file requires a sequence of steps involving handshaking between the Open Counter and the Open Manager. The goal is to ensure that there is no chance of a file descriptor being lost due to some error while opening the file.

First here is the signature for the Open Counter manager.

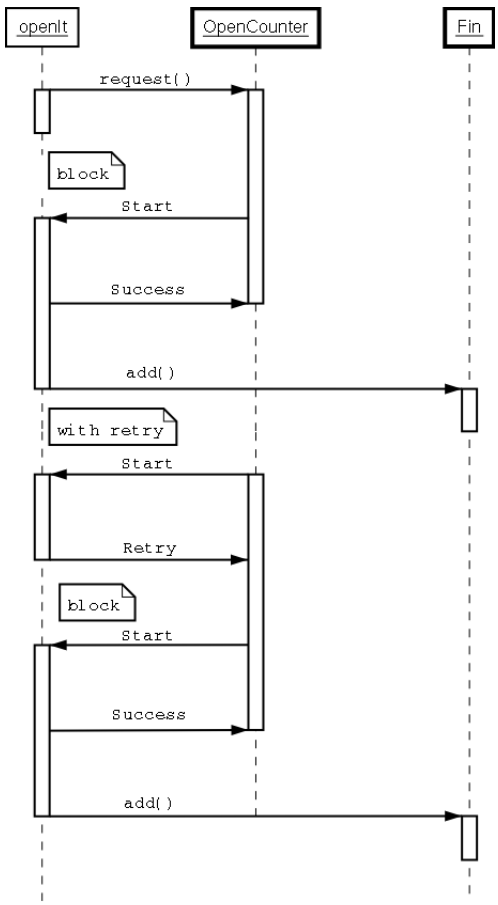
```
signature OPEN_COUNTER =
sig

    (*  This represents some number of file descriptors. It ensures
        that a release matches the allocation.
    *)
    type Allocation

    datatype Response =
            Success
        |   Fail of Allocation
        |   Retry of Allocation

    (*  Return the response on the supplied channel. *)
    type Start = Allocation * Response CML.chan

    (*  Pass in a channel to receive the start message. *)
    val request:   (int * Start CML.chan) -> unit

    (*  Release n file descriptors. *)
```

```
    val release:    Allocation -> unit

    (*  Return the number open and the number pending. *)
    val stats:      unit -> int * int
end
```

An `Allocation` value represents some number of open files. It provides some protection against programming errors by ensuring that the client can only return exactly the same number that it allocated.

The file allocation starts with a call from the `openIt` function of the Open Manager to the `request` function of the Open Counter. The Open Counter will, either immediately or some time later, start a handshake with the Open Manager using the channel in the request. The `Start` type is the message passed to the Open Manager to start the handshake. The Open Manager attempts to open the files and responds with a `Response` type. There is provision for retrying an open that fails due to an unexpected lack of file descriptors. In this case the Open Manager will go to the end of the queue to wait for more descriptors to become available. Figure 9-3 shows the sequence of the handshake for the Success and Retry cases. `Fin` is the specialised finaliser within the Open Manager.

Figure 9-3. The File Opening Handshake.



A consequence of this design is that only one open operation can occur at a time. The Open Counter runs the handshaking sequentially. This shouldn't be a problem since file opens, even the forking of CGI-BIN scripts, are quick. If this sequential processing proves to be a problem the Open Manager could be changed to run concurrent handshaking.

Here is the code for the `openIt` function.

```
fun openIt abort arg =
let
    val schan = CML.channel()

    (*  We may have to try several times.

        To be safe from deadlock there must be no possibility
        of an exception preventing the state transitions from
        completing. Otherwise the counter will block forever.

        So when we abort we must leave a thread behind to finish
        the handshaking. Trying to remove the pending request from
        the counter risks race conditions.
    *)
    fun try() =
    let
        fun got_alloc (alloc, rchan) =
        (
            case Impl.openIt arg of
              Impl.Success opn => (CML.send(rchan, Ctr.Success);
                                   SOME (opn, alloc))

            | Impl.Fail =>  (CML.send(rchan, Ctr.Fail alloc); NONE)

            | Impl.Retry => (CML.send(rchan, Ctr.Retry alloc); try())
        )
        handle _ => (CML.send(rchan, Ctr.Fail alloc); NONE)


        fun got_abort() =
        let
            fun dummy() =
            let
                val (alloc, rchan) = CML.recv schan
            in
                CML.send(rchan, Ctr.Fail alloc)
            end
        in
            CML.spawn dummy;
            NONE
        end
    in
        CML.select[
            CML.wrap(CML.recvEvt schan, got_alloc),
            CML.wrap(Abort.evt abort, got_abort)
```

```
            ]
    end
in
    (*  Start trying *)
    Ctr.request (Impl.num_fds, schan);

    (*  Once opened, set up a finaliser on the Opened value. *)
    case try() of
      NONE     => NONE
    | SOME farg => SOME(Fin.add farg)
end
```

A Specialised Open Manager

Here is the code the creates `TextIOReader`, an Open Manager specialised to reading text files using the `TextIO` module (which has the handy `inputLine` function).

```
local
    structure E  = Posix.Error
    structure TF = TextFrag

    structure Impl =
    struct
        val     name = "TextIOReader"
        type    Arg = string
        type    Opened = TextIO.instream
        type    Closed = unit

        val     num_fds = 1

        datatype Result =
                Success of Opened
            |   Fail
            |   Retry


        fun openIt file =
        (
            Success(TextIO.openIn file)
        )
        handle
          x as IO.Io {cause = OS.SysErr (_, SOME err), ...} =>
```

```
        (
            if err = E.mfile orelse err = E.nfile
            then
                Retry
            else
            (
                Log.logExn x;   (* a real error *)
                Fail
            )
        )
        | x => (Log.logExn x; Fail)


        fun closeIt strm =
        (
            TextIO.closeIn strm
        )
        handle x => Log.logExn x

    end
in
    structure TextIOReader = OpenMgrFn(structure Impl = Impl)
end
```

This code appears at the top module level. It defines two modules and an alias E for the `Posix.Error` module and the usual `TF` alias. The `E`, `TF` and `Impl` are made private to the TextIOReader module using `local...end` syntax.

The `Impl` module has to include all of the declarations in the `OPEN_MGR_IMPL` signature, including the `Result` type, which varies with the `Opened` type. What it opens is a file path so `Arg` is a string. The opened value is an input text stream. This only requires one file descriptor. The `openIt` function checks for the `EMFILE` and `ENFILE` error codes which indicate that no file descriptor is available. The open will be retried later in this case.

Since the web server is going to be forking/exec-ing CGI-BIN scripts it should be setting the close-on-exec flag on most of the files that it opens. It would be a security breach to let scripts inherit internal files, sockets, etc.

Unfortunately there is no mechanism in SML/NJ to operate on a `TextIO` or `BinIO` stream at the file descriptor level. I do do it for sockets though.

The Signal Manager

The web server should be catching signals so that it can clean up temporary files and such when it is killed. A CML thread can't be interrupted but I can broadcast an interrupt message to all interested threads. But this is rather awkward to handle. Each thread would have to be listening for an interrupt at each place where it may block for a while. At the moment all that I do is terminate the server by calling the common `fail()` function. This shuts down CML and I have registered a shutdown handler in the Main module. This handler can clean up for the server. See the section called *The Startup Module*.

I also need to distribute a signal indicating when a garbage collection has been done. This is used by finalisation code, for example see the section called *Finalisation*.

Signal handling is provided by the `Signals` module, see the section called *Signals* in Chapter 4. The GC signal is broadcast to the server using the `Multicast` module of the CML library. To use this you create a channel to carry a message stream and any thread wishing to receive these messages creates a port which listens to that channel. The signature for the signal manager is

```
signature SIGNAL_MGR =
sig

    (*  Each client must have its own port. *)
    type GcPort
    type IntPort

    datatype Interrupt = SIGINT | SIGTERM

    (*  This sets up the signal handling. *)
    val init:   unit -> unit
```

```
    (*  Create a new client port.
    *)
    val mkGcPort:   unit -> GcPort
    val mkIntPort:  unit -> IntPort


    (*  This creates an event for the arrival of the
        next GC signal. Call it anew for each GC.
    *)
    val gcEvt:  GcPort -> unit CML.event


    (*  This creates an event for the arrival of the
        next interrupting signal.
    *)
    val intEvt: IntPort -> Interrupt CML.event

end
```

GC messages don't carry any information so I just use the unit type. They are delivered to a `GcPort`. The `mkGcPort` function creates a new port to receive a GC message. The `gcEvt` function returns an event that a thread can select on. The code for handling the GC signal is

```
type GcPort  = unit Multicast.port
type IntPort = Interrupt Multicast.port

val gc_mchan: unit Multicast.mchan option ref = ref NONE
val int_mchan: Interrupt Multicast.mchan option ref = ref NONE


fun init() =
(
    gc_mchan  := SOME(Multicast.mChannel());
    int_mchan := SOME(Multicast.mChannel());

    Sig.setHandler(Sig.sigGC, Sig.HANDLER gc_handler);
    Sig.setHandler(Sig.sigINT, Sig.HANDLER int_handler);
    Sig.setHandler(Sig.sigTERM, Sig.HANDLER int_handler);

    (*  We'd better catch this for when writing to sockets. *)
    let
        val s = valOf(Sig.fromString "PIPE")
```

```
    in
        Sig.setHandler(s, Sig.HANDLER pipe_handler)
    end;
    ()
)


and gc_handler(_, _, kont) =
(
    Log.testInform Globals.TestTiming Log.Debug
        (fn()=>TF.S "GC signalled");
    Multicast.multicast(valOf(!gc_mchan), ());
    kont
)
```

The GC signal handler just broadcasts a message and continues the server.

The channels can't be set up until the server is running with the CML library. So an `init` function is required to set up the channels. This must be called at the very beginning of the server startup since the open file manager (the section called *The Open File Manager*) requires it before any files can be opened. The test harnesses must do the same.

The Singleton Module

This module is a simple encapsulation of the start-up of a thread that implements a singleton concurrent object. The object is represented by a CML channel or mailbox that receives the messages of its API.

The module is a functor that is specialised by the type of the input channel, a function to create the channel and the function that runs in the thread. Here is the complete code for the module.

```
functor Singleton (
    type input
    val  newInput: unit -> input
    val  object: input -> unit -> unit
    )
    : SINGLETON =
struct
```

```
    structure SV = SyncVar

    type input = input

    val input: input option ref = ref NONE

    (*  An initialised mvar can be saved over an exportML.
        The value it contains is the baton, like a binary semaphore.
    *)
    val mutex = Mutex.create()


    (*  The double-checked locking will be safe in CML since it
        isn't really multi-tasking or SMP (cf Java).
    *)
    fun get() =
    (
        case !input of
          NONE =>
            let
                fun init() =
                (
                    case !input of
                      NONE =>
                        let
                            val i = newInput()
                        in
                            input := SOME i;
                            ignore(CML.spawn (object i));
                            i
                        end
                    | SOME i => i
                )
            in
                Mutex.lock mutex init
            end

        | SOME i => i
    )
end
```

The module provides a single function called `get` which returns the channel to the object. A thread for the object is spawned the first time that the `get` function is called.

The channel is stored in a static variable so its update must be synchronised to protect against more than one thread calling the `get` function at the same time. See the section called *The Mutex Module*.

Here is an example of the use of this module. This is taken from the section called *The Log Module*.

```
structure Logger = Singleton(
                        type input    = LogRequest Mailbox.mbox
                        val  newInput = Mailbox.mailbox
                        val  object   = log_server
                        )

fun send_request req = Mailbox.send (Logger.get(), req)
```

The Text Module

A common operation in the server is constructing text messages. This varies from constructing the headers of HTTP responses through to constructing error messages for logging. In a traditional language like C a programmer typically assembles a message into a buffer by copying text fragments. This is copy-by-value for the fragments. If you try to be more efficient and do copy-by-reference for the fragments in C you can easily end up with slower code. This is because you will probably end up calling `malloc` a few times and the overhead will probably outweigh the cost of copying most strings. The memory management issues make it worse. Strings are often copied just to isolate the various domains of ownership of memory and also to protect against strings being modified.

The low-overhead memory allocation of SML/NJ changes the balance in favour of copy-by-reference. A list of strings is a list of pointers to strings and can usually be constructed faster than the strings can be copied and of course memory management is not an issue.

The TextFrag module implements a data structure that represents a string as an aggregate of string fragments[1]. It is also independent of the different line termination conventions, a LF or a CR-LF. Here is the `Text` type.

```
datatype Text =
        Empty
    |   Nl                (* a line break, perhaps CRLF *)
    |   WS                (* exactly one blank character *)
    |   S of string
    |   L of string list  (* concatenation of some strings *)
    |   C of Text list    (* concatenation of texts *)
```

In the following description I abbreviate the module name to `TF` (as I do in the server code). The `TF.S` constructor is the simplest case that represents a single string. The string should not contain any new-line character. Use the `TF.Nl` constructor to separate lines. This will be substituted later with whatever line-termination convention you want. The TF.C constructor combines fragments together. For example here are two lines of text.

```
val foxes =
    TF.C [TF.S "The quick brown fox", TF.Nl,
          TF.S "jumps over the lazy dog"]
```

The `TF.WS` constructor is available as a useful separator when joining fragments. The `TF.L` constructor handles the common case of concatenating multiple strings.

Here is the main part of the API.

```
datatype LineSep = UseLf | UseCrLf

(*  This applies the function to each string piece. The function
    could be print() for example.
*)
val apply:  LineSep -> (string -> unit) -> Text -> unit

(*  Calculate the length in characters of the text.
*)
val length: LineSep -> Text -> int

(*  This is like apply but it prints the prefix before
    each subsequent line.
*)
val appPrefix:  string -> (string -> unit) -> Text -> unit

(*  Produce the string that the Text corresponds to.
*)
```

```
val toString:  LineSep -> Text -> string
```

The `apply` function can be used to print a text fragment. For example to print the foxes fragment above to the standard output:

```
TF.apply TF.UseLf print foxes
```

The `appPrefix` is similar but inserts a prefix string before subsequent lines and only terminates with LF. I use this for indenting continuation lines in error messages. The `toString` function concatenates all of the text fragments by copying, which you sometimes may have to do.

I'll just show the implementation of the `apply` API to show you how it works. The other functions are similar. Note that `app` is the standard SML `List.app` function.

```
(*  crlf is the string to apply in place of Nl. *)
fun applyP crlf func Empty   = ()
|   applyP crlf func Nl      = func crlf
|   applyP crlf func WS      = func " "
|   applyP crlf func (S s)   = func s
|   applyP crlf func (L ss)  = app func ss
|   applyP crlf func (C lst) = app (applyP crlf func) lst


fun lsep UseLf   = "\n"
|   lsep UseCrLf = "\r\n"

fun apply sep func text = applyP (lsep sep) func text

fun appPrefix prefix func text = applyP ("\n" ^ prefix) func text
```

A complex example of text fragments can be found in the directory fancy indexing code that is described in the section called *The Directory Node Handler*.

The TmpFile Module

In the section called *System Resource Management* in Chapter 8 I described how the server must manage the amount of disk space used by temporary files. If there is insufficient disk space for a connection to save an incoming entity then the connection must be blocked until the space is available or the connection times-out. If a connection is aborted then any temporary files that belong to it must be deleted.

At the moment the only use I have for temporary files is for saving incoming entities in HTTP requests. I label the body file that is associated with a connection with the port number for the connection. Then when a connection is closed it can be easily found and deleted. I don't use the finalisation facility of the section called *Finalisation* since I want disk files to be cleaned up as soon as possible and I can rely on the connection code to catch all error or abort conditions for the connection.

Here is the API for the `TmpFile` Manager.

```
type TmpFile

(*  Allocate a new file for the given port number.
    If we give up trying to create the file then we return NONE.
*)
val newBodyFile: Abort.Abort -> string -> int -> int -> TmpFile option

(*  Get the file name. *)
val getName:        TmpFile -> string

(*  This releases the files associated with the port number.
    They will be deleted.
*)
val releasePort:    int -> unit


(*  This sets the temp file disk space limit. It must be
    called before any temp files are created, preferably
    from the config.  The size is in bytes.
    The limit must be no larger than 10^9 bytes.
*)
val setDiskLimit:   int -> unit
```

The `TmpFile` type represents the allocation of a name and disk space for the file. The caller must still write the data into the file. The `newBodyFile` function allocates a `TmpFile` for saving the entity body from a HTTP request. The arguments are the abort condition, temporary directory name, file length and port number. Only one body file should be allocated with the same port number.

The `getName` function will return the full path of the file for the caller to write to. The `releasePort` function deletes all files that are labelled with the given port number. The caller should not retain any `TmpFile` values with that port number after doing this.

The `setDiskLimit` function sets the number of bytes that are available for sharing out to the temporary files. I assume that all files will be in the same temporary directory or if not then they are all in the same file system. The file system must not be mounted over NFS since files are created with exclusive locking. This is specified in the section called *The Server Parameters* in Chapter 8.

The manager is a singleton concurrent object. Here are the types in the module.

```
datatype TmpFile = TmpFile of {
        id:         int,
        port:       int,
        file:       string,         (* absolute file path *)
        len:        int             (* a size estimate *)
        }

and AllocRequest =
        Record of Pending
    |   Release of int              (* release all files on the port *)
    |   Undo of TmpFile             (* undo one allocation *)


and Reply = Success of TmpFile | Reject

(*  This state could allow multiple files on the
    same port number.  It must be pure to allow
    recursion through allocate().
*)
and State = State of {
```

```
        tmps:       TmpFile list,   (* successfully allocated *)
        pending:    Pending list,
        used:       int,            (* disk space used in bytes *)
        id_cnt:     int,            (* to allocate ids *)
        last_warn:  Time.time option(* time of the last warning *)
        }

(*  A pending request has the port, file and length
    and a reply channel.
*)
withtype Pending = int * string * int * Reply Sy.ivar
```

The `TmpFile` value includes a unique identifier for simple equality testing. `AllocRequest` and `Reply` make up the protocol for the Manager. The Undo request is for cleaning up if a request is aborted. The Manager's state includes a list of all allocated `TmpFile` values, all pending requests and counters for disk space and identifiers. The Manager will log a warning if it runs short on disk space. These warnings are limited to 1 per second to avoid flooding the log. The `last_warn` field records the time when the last warning was given.

The disk space limit is recorded in a static variable.

```
val disk_limit = ref (valOf Int.maxInt)

fun setDiskLimit n = (disk_limit := n)
```

This is set once when the configuration file is read (see the section called *The Config Module - Interface*) so I don't have to worry about concurrent access to the variable.

Here is the `allocate` function that implements the `Record` request from the `newBodyFile` function.

```
and allocate
        (state as State {tmps, pending, used, id_cnt, last_warn})
        (pend as (port, file, len, rvar))
        =
let
    val _ = Log.testInform Globals.TestTmpFile Log.Debug
                (fn()=>TF.L ["TmpFile allocate file ", file])
in
```

```
    if used + len <= !disk_limit
    then
        let
            val tmp = TmpFile {
                        id      = id_cnt,
                        port    = port,
                        file    = file,
                        len     = len
                        }
        in
            Sy.iPut(rvar, Success tmp);

            State {
                tmps    = tmp::tmps,
                pending = pending,
                used    = used + len,
                id_cnt  = id_cnt + 1,
                last_warn = last_warn
                }
        end
    else
        let
            val now = Time.now()
        in
            if last_warn = NONE orelse
               Time.toMilliseconds(Time.-(now, valOf(last_warn)))
                   >= 1000
            then
                Log.log Log.Warn
                    (TF.S "TmpFile: Tmp disk space limit exceeded")
            else
                ();

            State {
                tmps    = tmps,
                pending = pend::pending,
                used    = used,
                id_cnt  = id_cnt,
                last_warn = SOME now
                }
        end
end
handle _ => (* e.g. integer overflow *)
    (
        Sy.iPut(rvar, Reject);
        Log.testInform Globals.TestTmpFile Log.Debug (fn() => TF.L [
            "TmpFile allocation error on port ", Int.toString port]);
```

```
        state
    )
```

This just does a simple check of the requested space against the amount available. If there is enough it sends a `Success` reply. If there isn't enough space then the request is just added to the pending list and a warning may be logged.

I must be careful not to let an exception abort the Manager. The (warn level) logging and `FileIO.removeFile` functions do not raise exceptions. But I have to watch out for integer overflow with the large numbers that file sizes might be.

The `release` and `undo` functions implement the remaining two requests to the Manager.

```
(*  Remove all those files on the port. *)
and release state the_port =
let
    fun keep (TmpFile {port, ...}) = (the_port <> port)
in
    remove state keep
end



(*  Remove based on the id. *)
and undo state tmp =
let
    val TmpFile {id = tmp_id, ...} = tmp
    fun keep (TmpFile {id, ...}) = (tmp_id <> id)
in
    remove state keep
end
```

These are just two different ways to remove files. The first removes files on a given port. The second removes a particular file based on its unique id. The common code is in the `remove` function. The different behaviour is represented by the `keep` functions which decides which files to keep.

```
and remove
        (state as State {tmps, pending, used, id_cnt, last_warn})
```

```
        keep
        =
let
    val _ = Log.testInform Globals.TestTmpFile Log.Debug
                (fn()=>TF.S "TmpFile removing")

    (*  First remove files.  Calculate the new used space.
        The pending list is separated out.
    *)
    fun filter [] new_tmps new_used =
    (
        State {
            tmps    = new_tmps,
            pending = [],
            used    = new_used,
            id_cnt  = id_cnt,
            last_warn = last_warn
            }
    )
    |   filter (tmp::rest) new_tmps new_used =
    (
        if keep tmp
        then
            filter rest (tmp::new_tmps) new_used
        else
            let   (* This raises nothing. *)
                val TmpFile {file, len, ...} = tmp
            in
                FileIO.removeFile file;
                filter rest new_tmps (new_used - len)
            end
    )


    (*  Retry all of the pending requests. Any that can't
        be satisfied will end up in the pending list again.
        We use a FIFO order for rerunning the requests.
    *)
    fun retry []        new_state = new_state
    |   retry (p::rest) new_state = retry rest (allocate new_state p)

    val filtered_state = filter tmps [] used
    val final_state = retry (rev pending) filtered_state
in
    final_state
end
```

The `keep` function is used to filter the `TmpFile` values that the Manager has retained. If a file is deleted then the amount of used space is reduced by the file's length. There may then be space for some of the pending requests to be satisfied. The simplest way to handle this is to rerun all of the pending requests by feeding them to the `allocate` function again. If there are still pending requests that can't be satisfied then they will end up back in the `pending` list in the state. Note that since the pending requests are pushed onto the front of the list but I want to serve them in first-come-first-served order I need to reverse the pending list before rerunning it. Since there should be few pending requests if any in normal operation this design will be efficient enough.

The processing of the `newBodyFile` function has two parts. The first part is to allocate a file name. I choose names of the form "portnnnn" where "nnnn" is the port number. If there is a name clash then subsequent names of the form "portnnnn_1" etc. are tried. Here is the code for the first part.

```
fun newBodyFile abort tmp_dir len port =
let
    (*  If we get a name clash then add _ suffixes.
    *)
    fun try n =
    let
        val base = concat[
                    "port", Int.toString port,
                    if n = 0 then "" else ("_" ^ (Int.toString n))
                    ]

        val file = Files.appendFile tmp_dir base

        val () = Log.testInform Globals.TestTmpFile Log.Debug
                    (fn() => TF.L ["newBodyFile trying ", file]);
    in
        if FileIO.exclCreate file
        then
            allocate port file len
        else
        (
            if n > 10
            then
            (
                Log.error ["File name clashes for file ", file];
```

```
                NONE                (* give up *)
            )
            else
                try (n+1)
        )
    end
```

The `try` function makes around 10 attempts to create the file. The `FileIO.exclCreate` function ensures that the name is exclusively allocated to the server. There may be left-over files using the same port if a previous run of the server crashed.

The second part is the reservation of disk space. This is done with the following `allocate` function in `newBodyFile`.

```
and allocate port file len =
let
    val rvar = Sy.iVar()


    fun got_reply (Success tmp) = SOME tmp
    |   got_reply Reject        = (FileIO.removeFile file; NONE)


    and got_abort() = (CML.spawn dummy; NONE)


    (*  Run this to catch left-over allocation requests. *)
    and dummy() =
    (
        case Sy.iGet rvar of
          Success tmp => CML.send(Alloc.get(), Undo tmp)
        | Reject      => ()
    )
in
    CML.send(Alloc.get(), Record(port, file, len, rvar));

    CML.select[
        CML.wrap(Sy.iGetEvt rvar, got_reply),
        CML.wrap(Abort.evt abort, got_abort)
        ]
end
```

The function sends a Record message to the Manager to record the file's information. Then it waits for the reply or an abort condition. If a `Success` reply is returned then the TmpFile value is returned to the caller. If a `Reject` reply is returned then there has been some kind of error. The file is removed and `NONE` is returned to the caller.

If there is an abort condition then I need to abort the request somehow. It would be tricky to try to extract the request out of the Manager without race conditions. The CML `withNack` function could be used to inform the manager about the loss of the client. But for simplicity I like to leave the request with the Manager and instead leave behind a dummy receiver in a new thread that will immediately undo a successful allocation. The `Undo` message will remove the file. Assuming that aborts are rare this will be efficient enough.

The URL Module

This module implements a parsed representation for a URL. For details on the URL syntax see the section called *URL Syntax* in Chapter 8.

```
datatype URL = HTTP_URL of {
    host:       string option,
    port:       int option,
    userinfo:   string option,      (* user name/password *)
    path:       URLPath,
    query:      string option,      (* in the undecoded form *)
    fragment:   string option       (* # suffix, undecoded *)
    }

and URLPath = URLPath of {
    segs:       Segment list,
    absolute:   bool
    }

and Segment = Segment of {
    part:       string,
    params:     string list
    }
```

All of the parts in the URL are separated into fields in the URL type. I only support HTTP URLs at the moment. The datatype allows room for expansion. All of the HTTP URL syntax is supported but this is more general than the server actually needs. For example the syntax allows parameters to be attached to each segment in the URL path but you would only expect to find parameters on the last segment in practice. The server only ever expects to encounter absolute URL paths and never fragments.

I will only describe the API. The code is a lot of long-winded string manipulation to break the URL at all the different kinds of delimiters. Here is the signature for the API.

```
exception BadURL of string          (* with a reason *)

val emptyURL:   URL


(*  This parses a general URL.  It raises BadURL if the syntax
    is not parsable as a HTTP URL.
*)
val parseURL:   string -> URL


(*  This parses just the path part of a URL, excluding the fragment.
    It raises BadURL if the syntax is not parsable.
*)
val parseURLPath:   string -> URLPath


(*  This parses just a simple path which contains no parameters.
    It raises BadURL if the syntax is not parsable.
*)
val parseSimplePath:   string -> URLPath


(*  This tests if two paths match, ignoring parameters. *)
val samePath:   URLPath -> URLPath -> bool

(*  This splits a path into a prefix of part names and a final name.
    E.g. /a/b/c becomes SOME ([a,b], c) and / becomes NONE
*)
val splitPath:  URLPath -> (URLPath * string) option

(*  Convert back to a valid URL in string form.
```

```
    This introduces escapes etc.  For now we only escape the
    "reserved" character class.  We could also escape the mark
    characters for safety.  Netscape does.

*)
val URLToString:    URL     -> string
val pathToString:   URLPath -> string

(*  This removes the % URL escapes from the string.
    Bad escapes are passed through unchanged.
*)
val unescapeURL:    string -> string

(*  This escapes anything that is not an unreserved character in
    the string.
*)
val escapeURL:      string -> string
```

The `parseURL` function is the main interface. If the URL has an invalid format then the `BadURL` exception is raised. This exception carries an error message that the caller may choose to log. The `parseURLPath` function is not currently used. The `parseSimplePath` function is for URL paths in the node sections of the configuration file (see the section called *The Node Parameters* in Chapter 8). A simple path does not allow parameters on a path segment.

The `URLToString` function is the opposite of parseURL. It is used when formatting headers and also in generating HTML. It must reintroduce escaping for unsafe characters. The `escapeURL` and `unescapeURL` functions take care of the escaping. The remaining functions are utilities of occasional use.

Notes

1. It was inspired by the `wset` type in ML Server Pages from Moscow ML

Chapter 10. Conclusion

The aim of this book has been to demonstrate that functional languages, in particular SML, can be used to build the kind of real-world projects that you might ordinarily only think of using C or C++ for. In the following I will discuss some of the lessons learned from developing the web server project in SML/NJ. I'll also point you at some related languages to look into.

SML/NJ vs Real-World Needs

To tackle real-world projects a language and its environment needs to:

- scale to large quantities of code;
- have the performance to handle real-world loads;
- interface with infrastructure such as databases, graphics etc.

Large-scale Development

Functional languages certainly scale to the size of real-world projects by nature. The productivity improvement from a functional language allows a programmer to tackle much larger projects. I estimate at least a 10 to 1 improvement in productivity over the C language just from a reduction in the number of lines of code needed. There is a large productivity boost from the absence of memory management bugs and the like that plague C programs. Further the strong static type system often means that sizable pieces of SML code just works first time.

The SML module system goes a long way towards managing large-scale code. (But its inability to handle groups of mutually dependent modules is a minus).

Performance

Over the years computer scientists have developed many "very high level" languages to help solve the "programming crisis" and improve programming productivity. In the early stages of research they are usually interpreted rather than compiled and the implementation is slow. Often they don't progress beyond the research stage. This leaves the impression that advanced languages are by nature slow and not suitable for real-world use. Unfortunately programmers are prone to myths about programming languages and seldom revisit them.[1]

My experiments with the Swerve server in this book show that SML/NJ does perform well. Without much trouble it achieved around 2/3 the speed of Apache, which is written in C. Anecdotal evidence has it that C++ is about 1/2 the speed of C. The speed drop is mainly due to the excessive copying encouraged by copy constructors and the overhead of virtual method calls. Even a good Java implementation is slower again. This suggests that SML/NJ can certainly compete with C++ or Java.

Having said that, there are still some performance issues that need to be noted. In the section called *Basic SML/NJ Performance* in Chapter 7 I ran some basic speed tests. The performance on inner loops and low-level byte handling could do with some improvement. This speed loss is partially compensated for by the much better heap performance. Other implementations of SML such as the MLton compiler[*MLton*] work harder at optimisation.

Infrastructure

The big impediment to wider use of SML in the real-world is support for features such as databases, graphics, encryption, native languages (unicode), true concurrency etc.

The standard basis library is looking rather dated. It hasn't progressed in years. A future release of SML/NJ will include a native C-language interface.

This will allow your SML program to dynamically link to a shared library and call its C API. It includes a tool that generates SML stubs for all of the API functions found in the library's header files. This will make it possible to call libraries for database access, file encryption and compression etc. But I think that the basis library will need further work to integrate this properly. Personally I would like to see a plug-in mechanism for the SML/NJ run-time to make it easier to extend.

The choice of a web server for this book neatly avoids the issue of database programming. Web servers typically delegate this to CGI scripts which you could write in Perl or PHP. SML/NJ desperately needs an ODBC-like mechanism.

This book has been silent too on the matter of graphics programming. Image handling (e.g. JPEG, PNG) needs access to standard libraries as described above. To develop a GUI with SML/NJ you have a choice of the eXene system, supplied with SML/NJ, or the SML/Tk package[*SMLTK*]. I haven't used eXene. It's an experiment in designing a concurrent GUI toolkit for CML. But I would prefer a more conventional look and feel such as Tk. But SML/Tk is not written for a concurrent environment. Personally I consider Concurrent ML to be a "killer" feature of SML/NJ. A GUI should be multi-threaded for responsiveness. It's a shame that CML is not built-in to SML/NJ instead of being an add-on.

Related Languages

SML/NJ is not the only functional language I could have chosen to demonstrate real-world programming.

The OCaml system[*OCaml*] is an implementation of a language in the ML family. The language is different from Standard ML and a bit on the experimental side for my taste. However it has recently seen much active development in the area of infrastructure[*OCamlTools*]. For example for

graphics it has interfaces to Tk, Gtk+ and OpenGL. There are interfaces for the PostgreSQL and MySQL databases. If you're wondering why I didn't choose OCaml for this book, it's that much of this progress has been made while I've been writing this book. You should certainly look into it.

An alternative in the lazy functional language arena is Haskell[*Haskell*]. Haskell compiles to machine code but its lazy evaluation tends to make it run slower than SML/NJ. There is good infrastructure support especially if you are programming on Microsoft Windows. It talks COM and CORBA. For graphics there are interfaces for Tk and Gtk+. For databases there are interfaces for PostgreSQL and MySQL and on Windows, ODBC.

To Finish

The bottom line is: yes you can nowadays develop real-world applications in functional languages, getting leverage off of their special features such as greater productivity and reliability. Try it out!

Notes

1. Here are some common ones. *Lisp is slow.* Optimising compilers have been available for years. *Ada is huge and bloated*. What, C++ isn't? *C must be efficient because you can program down to the bare metal*. Nope, its 1970s-vintage machine model no longer fits well with modern computers.

Appendix A. Learning SML

In this appendix I describe some of the resources available for learning SML and SML/NJ in particular. The Standard ML of New Jersey home page [*SML*] has links to these and more.

Books

The SML/NJ home page recommends two books: *ML for the Working Programmer*[*Paulson*] and *Elements of ML Programming*[*Ullman*]. Both of these address the 1997 revision of the language[*Milner*]. Earlier books only cover the 1990 definition..

Of the two I've only read the Paulson book. It is aimed at the more experienced programmer. It's major worked programming example is a simple mathematical theorem prover. This is based on the author's experience with the Isabelle theorem prover (see [*SMLProjects*]). You'll also find some focus on other top-shelf subjects such as lambda calculus and formal reasoning about programming.

The Ullman book is an introductory book on programming with SML which assumes no previous knowledge of functional programming. It focuses on using the SML/NJ implementation.

Robert Harper has had tutorial notes on SML available for a number of years. Recently he has expanded these into book form entitled *Programming in Standard ML*. At the time of writing you can download a draft copy of the book in PDF format from his home page at [*HarperHome*].

Tutorials

The tutorials available on the web vary a lot in their emphasis. The most

comprehensive is by Robert Harper which is now in book form, cited above.

The tutorial by Peter Lee[*LeeTut*] is a set of notes aimed at beginning students using the SML/NJ for the first time.

Mads Tofte has several sets of notes which emphasise using the module system of SML. You will need to follow the links from the SML/NJ home page to get these tutorials.

Andrew Cumming has a set of lessons entitled *A Gentle Introduction to ML*[*Cumming*]. These are very introductory.

Appendix B. Coping with the Compiler's Error Messages

Figuring out the error messages from the compiler can sometimes be the hardest part of getting a program to run. The syntax errors are easy enough but the type errors can be confusing even for experienced programmers. The good news is that once you get your program past the type checker there is often not much wrong with it. It's not unusual for programs hundreds of lines in length to just work the first time. The richness of the type system and its enforcement provide logic constraints that keep out many common programming errors.

Syntax Errors

The parser of the SML/NJ compiler uses error correction. This means that it attempts to correct a syntax error so that it can continue compiling. This lets it report more syntax errors in one run of the compiler. However the error messages report the corrections that it made and you have to invert its logic to figure out what your errors were.

The parser will attempt to either insert or delete one or more symbols to produce syntactically correct input. Here is an example.

```
fun f x
let
    val i = 1
in
    i * i
end

syn1.sml:1.7 Error: syntax error: inserting  EQUALOP
```

This says that on line 1 at character position 7 it inserted an equals symbol. This was the right correction. Here's a more notorious error.

```
structure Main =
struct
    fun f x =
    let
        val msg = "hello"
    in
        print msg;
        print "\n";
    end
end

syn2.sml:9.5 Error: syntax error: inserting  EQUALOP
```

The semicolon in a sequence of expressions is only allowed between the expressions. The one at the end of the last print is wrong. But the parser's correction is to insert something else after the semicolon to get it into an internal position. Curiously, if the function is not inside a structure the parser produces a slightly better error message: `syntax error found at END`. If you leave out the semicolon between the prints this is not a syntax error but you will get a type error.

Identifier Errors

In the terminology of SML an identifier is *bound* to some definition. If you haven't defined the identifier in an expression you will get a complaint about an unbound variable. If the identifier is in a binding pattern then you might have intended it to be a constructor from a datatype. Here are some examples.

```
fun f cmd =
(
    case cmd of
      [] => ""
    | (first::rest) => TxtIO.print frst
)

ubnd1.sml:5.24-5.35 Error: unbound structure: TxtIO in path TxtIO.print
ubnd1.sml:5.24-5.28 Error: unbound variable or constructor: frst
```

Record Errors

The binding patterns that select record fields introduce ambiguities that usually need some kind of explicit type constraint.

```
type Cmd = {
    name: string,
    data: int
    }

fun show {name, ...} = print name

fun process cmd = print(#name cmd)

fun size ({data, ...}: Cmd) = data

rec1.sml:6.1-6.34 Error: unresolved flex record
   (can't tell what fields there are besides #name)
type4a.sml:8.1-8.35 Error: unresolved flex record
   (can't tell what fields there are besides #name)
```

The binding pattern in `show` is called a "flex record". It can match against any record type that contains a field called `name`. But you aren't allowed to make a function polymorphic over all such record types so you have to constrain its type as done in the `size` function. The field selector `#name` is equivalent to the function `(fn {name, ...} => name)` so it has the same problem.

I prefer to use a datatype for all record types. The constructor will serve to constrain the type.

```
datatype Cmd = Cmd of {
    name: string,
    data: int
    }

fun show (Cmd {name, ...}) = name
```

Type Errors

The SML language is designed so that you can write your code without having to declare each variable with its type. Ideally you should be able to write no types anywhere and the compiler can figure out the type of each expression entirely by context. The language comes very close to this ideal. Only some ambiguities with overloaded numeric literals and record patterns spoil it.

Type expressions in SML can be quite complex. If a function is polymorphic then its type will feature type variables which will be tiresome to get straight. Having the type checker figure them out for you is a big advantage.

The disadvantage of this design is that when there is a type error the cause of the error can be very obscure. Where an identifier is used in several locations in your program the type checker compares each location to see if the use is consistent. For example if you use an identifier as the argument to a function that takes a string and also to one that takes an integer then the identifier can't be both a string and an integer. You will have to decide which one is wrong. If you use the identifier in a great many locations you may have to inspect all of them to find out which one is incorrect.

When the type checker is studying your program it reads it from top to bottom and decides on the type of an identifier from the first location it encounters that supplies it with decisive information. Every following location is checked against this type and if there is a mismatch then an error is reported. If it happens that the first location is the wrong one then all of the remaining locations will report errors.

The message that is generated for each type error will typically contain an abstract of the offending source code and a report of two type expressions that didn't match. Usually the code is a function call and the mismatch is between the expected type of the argument and the actual type of the argument expression. To figure out the type error you have to compare the two type expressions. They often contain internal type variables written like `'Z`. A type variable will match with any type. Type variables with the same letter in the same type expression must be the same type.

Sometimes you will reach a point where the type checker insists that there is an error at some location and you are sure that it's not there but somewhere else. A good strategy is to put in an explicit type constraint to point out to the type checker what you think the type must be. The checker will then point out any other locations that don't match that type. You can put a type constraint on any expression, including literals.

The following sections show some typical examples and what went wrong in each case.

Simple Type Errors

The simplest error is an argument mismatch when the argument type is obvious.

```
fun f() = print 3

type1a.sml:1.11-1.18 Error: operator and operand don't agree [literal]
  operator domain: string
  operand:         int
  in expression:
    print 3
```

The message talks about an operator, the function, and an operand, its argument. The domain of an operator is the type that it expects. In this case it expected a `string` and was given an `int`.

Distinguishing the operator and operand is harder with curried functions.

```
fun f() =
let
    fun g(a, b) = a + b
in
    foldl g 0.0 [1, 2]
end

type1b.sml:2.1-6.4 Error: operator and operand don't agree [literal]
  operator domain: real list
  operand:         int list
  in expression:
```

```
    ((foldl g) 0.0) (1 :: 2 :: nil)
```

Here the operator is the expression `(foldl g 0.0)` which must take a list of reals for the final argument. The error is that a list of integers was supplied. We can surmise that lists in square brackets are represented internally in the compiler as applications of the list constructor operator `::`.

If you leave out a semicolon in a sequence expression you will usually end up with a type error. Here's a simple example.

```
fun f x =
let
    val msg = "hello"
in
    print msg
    print "\n"
end

type1c.sml:2.1-7.4 Error: operator is not a function [tycon mismatch]
  operator: unit
  in expression:
    (print msg) print
```

To the compiler it looks like you are passing the `print` function as an argument to the `(print msg)` expression. But this expression isn't even a function. Its type is `unit`.

If and Case Expressions

Some kinds of expressions don't end up in the error report how they started out in your code. Here's a silly example with two calls to the function `g`.

```
fun f lst =
(
    if length lst = 1
    then
        g print lst
    else
        g (fn s => (print s; print "\n")) (*lst*)
)
```

```
and g printer lst = app printer lst

type2a.sml:2.1-8.2 Error: types of rules don't agree [circularity]
  earlier rule(s): bool -> 'Z
  this rule: bool -> 'Y list -> 'Z
  in rule:
    false => g (fn s => (<exp>; <exp>))
```

The `if` expression becomes a `case` expression internally so that `(if b then x else y)` becomes `(case b of true => x | false => y)`. The source position in the message covers the range of the lines of the `if` expression.

The two cases are called rules. Each rule is treated like a function from the type of the case argument (here `bool`) to the type of the case result (here represented as the unknown type `'Z`). The type checker has a problem with the `else` part. The expression still needs an extra argument of type `'Y list` before it can return the case's type. This is because I forgot the `lst` argument.

The type checker uses the term "object" for the expression after the case keyword. Here is an example of a mismatch with the rules.

```
fun process (cmd: string) inp =
(
    case cmd of
      [] => []

    | (last::rest) =>
        (
            print last;
            app print rest;
            inp
        )
)

type2b.sml:3.5-11.3 Error: case object and rules don't agree [tycon mismatch]
  rule domain: string list
  object: string
  in expression:
    (case cmd
      of nil => nil
```

```
      | last :: rest => (print last; (app <exp>) rest; inp))
```

The rules clearly want a list of strings but there is an erroneous type constraint that says that `cmd` must be just a string.

Non-local Type Errors

Here's an example of a type error that propagates through a couple of levels of function call.

```
fun run() = print(process "stop")

and process cmd =
(
    case cmd of
      "go" => go()
    | _    => stop()
)

and go()   = (3, "done")
and stop() = (4, "stopped")

type3a.sml:1.1-12.28 Error: right-hand-side of clause doesn't
agree with function result type [tycon mismatch]
  expression:  int * string
  result type:  string
  in declaration:
    go = (fn () => (3,"done"))
type3a.sml:1.1-12.28 Error: right-hand-side of clause doesn't
agree with function result type [tycon mismatch]
  expression:  int * string
  result type:  string
  in declaration:
    stop = (fn () => (4,"stopped"))
```

The expected result type for the `go` and `stop` functions is determined to be `string` from the call to print in the first line. The error messages report the entire group of functions that are joined by the `and` keyword which doesn't localise the error much. If the `run` function comes last then the error is localised better.

```
fun process cmd =
(
    case cmd of
      "go" => go()
    | _    => stop()
)

and go()   = (3, "done")
and stop() = (4, "stopped")

fun run() = print(process "stop")

type3b.sml:11.13-11.34 Error: operator and operand don't agree [tycon mismatch]
  operator domain: string
  operand:         int * string
  in expression:
    print (process "stop")
```

Alternatively you can put a type constraint in a function declaration to break up the type propagation. This makes it clearer to the compiler and to anyone reading the code what is expected.

```
fun run() = print(process "stop")

and process cmd: int * string =
(
    case cmd of
      "go" => go()
    | _    => stop()
)

and go()   = (3, "done")
and stop() = (4, "stopped")

type3c.sml:1.13-1.34 Error: operator and operand don't agree [tycon mismatch]
  operator domain: string
  operand:         int * string
  in expression:
    print (process "stop")
```

Appendix C. Installation

In this appendix I describe how to download and install SML/NJ version 110.0.7 This is the latest stable version at the time of writing. You will be installing from the source files. You can get RPMs for Linux but these won't include the source, which is very useful. It's quite easy and fast to install it from source. My installation is on RedHat 7.1 Linux with the GCC 2.96 compiler.

A characteristic of the installation is that the installed files will end up in the same directory tree as the source files. So you must unpack the source in the place where you want the installed files to go.

To start, go to the SML/NJ home page. This is at

`http://cm.bell-labs.com/cm/cs/what/smlnj/index.html`

There is an FTP site at Sourceforge but I've found the Bell Labs site to be the most reliable. Follow the link "Downloading SML/NJ Software for Unix or Windows". This has a link to some on-line installation instructions.

1. Create a directory on your computer where you will install the SML/NJ system. The installation instructions call this `SMLROOT` and recommend a path like `/usr/local/sml`[1]. I recommend using a subdirectory for the version e.g. `/usr/local/sml/110.0.7` and a symbolic link `/usr/local/sml/current` to point to it. This will allow you to try out new versions. You use the `current` name in your CM files.
2. In the SMLROOT directory download the following files. These are for Intel x86 architecture on Unix. Use the FTP links. Don't download the Concurrent ML package from this page.

config.tar.Z	Installation scripts
runtime.tar.Z	Runtime system
cm.tar.Z	Compilation Manager

bin.x86-unix.tar.Z	for Intel Pentium processors running UNIX
smlnj-lib.tar.Z	Standard ML of New Jersey Library
ml-lex.tar.Z	ML-Lex lexical analyzer generator
ml-yacc.tar.Z	ML-Yacc parser generator
ml-burg.tar.Z	ML-Burg code-generator generator
sml-nj.tar.Z	Source code for SML/NJ compiler

You probably won't use ML-Burg but the installation script assumes that you have it. The compiler source file is useful for the source of the Basis library, which is in its `boot` subdirectory.

3. If you go back to the SML/NJ home page you can follow the links to the Concurrent ML page. In its "What's new" section there is a link to a directory containing a patched version. The link is

   `ftp://ftp.research.bell-labs.com/dist/smlnj/packages/cml`

   Download the `latest.tar.gz` file into the `SMLROOT` directory renaming it to `cml.tar.Z`. This will work as long as the `zcat` program on your computer is actually from the Gnu gzip package (which it is on Linux). The install script expects the `.Z` suffix.

4. In the SMLROOT directory unpack the `config.tar.Z` file.

   ```
   zcat config.tar.Z | tar -xf -
   ```

   This will create a subdirectory called `config`.

5. Edit the `config/targets` file to comment out the following line.

   ```
   TARGETS="$TARGETS eXene"
   ```

You won't need this for any of the examples in this book. You might be interested to read the eXene documentation so you could download the tar file later.

6. In the `SMLROOT` directory, NOT the `config` subdirectory, do

   ```
   sh config/install.sh
   ```

   This will compile and install everything. The tar files will be unpacked into a `src` subdirectory. The SML compiler is in the `bin` subdirectory. The CM files are in the `lib` subdirectory. You can delete the tar files.

   The compilation process first compiles the run-time. This is a normal C language compilation using `make`. I get lots of warnings from the `X86.prim.asm` file but this doesn't cause a problem.

   Next the compiler is built from its `bin` files. You will see lots of messages like `[Loading core.sml.bin]`. Then the various packages are compiled and installed. These produce the usual SML compiler blurb with lots of GC messages.

The `CM` files in the `SMLROOT/lib` directory are just aliases for files in the `SMLROOT/src` directory. The compiled binary files are saved in `CM` subdirectories in the source directories so you can't delete them. The aliases (and the configuration manager) use absolute paths so you can't move them. This rigidity has been addressed in later versions of the SML/NJ system.

You can update a package in the system, other than CM and the compiler, by just replacing the source directory with the new version and compiling in its source directory (where the `sources.cm` file is) in the usual way. With care you can make some changes to the run-time and recompile it without having to rebuild anything at the SML level. Look at the install script for the commands to make the run-time.

Notes

1. I use `/src/smlnj` and this path appears in all of my examples

Bibliography

[ACAPD] Carnegie Mellon University, *The Cyrus SML ACAP Server (http://asg.web.cmu.edu/cyrus/smlacapd/) (http://asg.web.cmu.edu/cyrus/smlacapd/)*.

[Allison] 1986, Cambridge University Press, *A Practical Introduction to Denotational Semantics*, Lloyd Allison.

[Appel1] 1994, Princeton University, *An Empirical and Analytic Study of Stack vs Heap Cost for Languages with Closures*, CS-TR450-94, Andrew W Appel.

[Appel2] 1992, Cambridge University Press, *Compiling with Continuations*, Andrew W Appel.

[AwkwardSquad] *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell (http://research.microsoft.com/~simonpj/papers/marktoberdorf.htm) (http://research.microsoft.com/~simonpj/papers/marktoberdorf.htm)*, Simon Peyton Jones..

[CLHTTP] Massachusetts Institute of Technology, *The Common Lisp Hypermedia Server (http://www.ai.mit.edu/projects/iiip/doc/cl-http/home-page.html) (http://www.ai.mit.edu/projects/iiip/doc/cl-http/home-page.html)*..

[CLU] *The CLU page (http://dmoz.org/Computers/Programming/Languages/CLU/) (http://dmoz.org/Computers/Programming/Languages/CLU/)*.

[CML] *The CML Home Page (`http://cm.bell-labs.com/cm/cs/who/jhr/sml/cml/index.html`) (http://cm.bell-labs.com/cm/cs/who/jhr/sml/cml/index.html)*.

[Clean] *The Clean Home Page (`http://www.cs.kun.nl/~clean`) (http://www.cs.kun.nl/~clean)*.

[CoolCrisp] *COOL : A Crisp object coordination language (`http://www.csc.uvic.ca/~mcheng/research/cool.htm`) (http://www.csc.uvic.ca/~mcheng/research/cool.htm)*.

[Cumming] *The A Gentle Introduction to ML (`http://www.dcs.napier.ac.uk/course-notes/sml/manual.html`) (http://www.dcs.napier.ac.uk/course-notes/sml/manual.html)*, Andrew Cumming.

[FoxNet] Carnegie Mellon University, *The Fox Project (`http://foxnet.cs.cmu.edu/HomePage.html`) (http://foxnet.cs.cmu.edu/HomePage.html)*.

[HarperHome] *The Robert Harper Home Page (`http://www-2.cs.cmu.edu/~rwh/`) (http://www-2.cs.cmu.edu/~rwh/)*.

[Haskell] *The Haskell Home Page (`http://www.haskell.org/`) (http://www.haskell.org/)*.

[HaskellServer] Sept 2000, Haskell Workshop, Montreal, Canada, *Writing High-Performance Server Applications in Haskell, Case Study: A Haskell Web Server (`http://www.haskell.org/~simonmar/papers/web-server.ps.gz`) (http://www.haskell.org/~simonmar/papers/web-server.ps.gz)*, Simon Marlow.

[Holub] 2000, apress, *Taming Java Threads*, Allen Houb.

[Hume] *HUME (Higher-order Unified Meta-Environment) (http://www-fp.dcs.st-and.ac.uk/hume/) (http://www-fp.dcs.st-and.ac.uk/hume/)*.

[LeeTut] *Using the SML/NJ System (http://www-2.cs.cmu.edu/~petel/smlguide/smlnj.htm) (http://www-2.cs.cmu.edu/~petel/smlguide/smlnj.htm)*, Peter Lee.

[Linda] *The Linda Coordination Language (http://www.cs.yale.edu/Linda/linda-lang.html) (http://www.cs.yale.edu/Linda/linda-lang.html)*.

[MLton] *The MLton compiler (http://www.sourcelight.com/MLton/) (http://www.sourcelight.com/MLton/)*.

[Milner] 1997, MIT Press, *The Definition of Standard ML (Revised)*, Robin Milner et al.

[OCaml] *The OCaml Home Page (http://caml.inria.fr/ocaml) (http://caml.inria.fr/ocaml)*.

[OCamlTools] *The OCaml Tools Page (http://caml.inria.fr/hump.html) (http://caml.inria.fr/hump.html)*.

[Okasaki] 1998, Cambridge University Press, *Purely Functional Data Structures*, Chris Okasaki.

[Paulson] 1996, second, Cambridge University Press, *ML for the Working Programmer*, L. C. Paulson.

[Reppy] 1999, Cambridge University Press, *Concurrent Programming in ML*, John H. Reppy.

[SML] *The Standard ML of New Jersey Home Page*
(`http://cm.bell-labs.com/cm/cs/what/smlnj/index.html`)
(http://cm.bell-labs.com/cm/cs/what/smlnj/index.html).

[SMLProjects] *The Projects that use SML*
(`http://www.cs.princeton.edu/~appel/smlnj/projects.html`)
(http://www.cs.princeton.edu/~appel/smlnj/projects.html).

[SMLTK] *The The SML/Tk Home Page*
(`http://www.informatik.uni-bremen.de/~cxl/sml_tk/`)
(http://www.informatik.uni-bremen.de/~cxl/sml_tk/).

[Simula] *The Simula page*
(`http://dmoz.org/Computers/Programming/Languages/Simula/`)
(http://dmoz.org/Computers/Programming/Languages/Simula/).

[Steele] 1990, Digital Press/Prentic Hall International, *Common Lisp*, Guy L Steele Jr.

[Ullman] 1998, Prentice-Hall, *Elements of ML Programming, ML97 Edition*, Jeffrey D. Ullman.

[WWWC] *World Wide Web Consortium (`http://www.w3.org/`)*
(http://www.w3.org/).

Glossary

Blurb

This is any text of little value or interest.

CM

This stands for the SML/NJ Compilation Manager. It also refers to the input files used by the manager.

CML

This is the Concurrent ML library. It adds operations for concurrent programming to SML/NJ.

continuation

A continuation is a virtual function that represents the rest of the computation of the program. By virtual I mean that it is behaves like a function that can be called from SML but underneath it just represents a transfer of control to other parts of the program.

copy by reference

When a data object is copied by reference the object itself is not duplicated. Instead a pointer to it is created. The copy and the original share the same memory locations. A change made to the original will result in the copy changing too.

See Also: copy by value.

copy by value

When a data object is copied by value the object itself is duplicated. The copy and the original occupy different memory locations. A change made to the original will not result in a change to the copy.

See Also: copy by reference.

currying

This refers to how a function with multiple inputs can be treated as a cascade of functions each called with a single input. For example the expression `(f a b)` means `(g b)` where `g = (f a)`.

datatype

This is a kind of type declaration in SML. It plays the role of enumeration and union types of C or C++.

dynamic

This refers to any information that cannot be known about a program until it is running. The information depends on the data it is processing.

See Also: static.

finalisation

This is a post-processing step applied to a data object after it has been collected by the garbage collector. For example a file object should be finalised by closing it since it can no longer be used by the program.

See Also: garbage collector.

functional programming

This style of programming proceeds by computing new values from existing values without changing any of the existing values.

See Also: imperative programming, pure function.

functor

This is a generic module in SML. It can be thought of as a function that generates a structure from one or more input structures.

See Also: generic, module, structure.

garbage collector

This is a part of the SML/NJ run-time that locates data objects that can no longer be used and makes their memory available for reuse.

generic

A piece of code is generic if it can be easily customised in the language to different kinds of input. This is similar to the idea of polymorphism but it handles a wider variety of kinds. The functor is the mechanism in SML for writing generic code. Templates are the corresponding mechanism in C++.

See Also: functor.

immutable

A variable is immutable if the value it represents can never change. This is normal for functional programs. In imperative programs variables are normally mutable.

See Also: functional programming.

imperative programming

This style of programming proceeds by altering values stored in variables.

See Also: functional programming, sneak path.

lex

This is the traditional lexical scanner generator available on Unix.

See Also: yacc.

live data

Any piece of data that can still be used by the program is called live. Dead data will eventually be collected by the garbage collector.

See Also: garbage collector.

mickey-mouse

A toy example.

module

In SML this refers to either a structure or a functor. The compiler compiles modules separately.

See Also: structure, functor.

monomorphic

A function is monomorphic if it only operates on data of one type. For example the `String.concat` function only concatenates lists of strings.

See Also: polymorphic.

polymorphic

A function is polymorphic if it performs the same operation on a family of types. For example the `List.length` function determines the length of any list without regard to the type of its elements.

See Also: monomorphic.

primitive

This refers to something at the lowest level of detail; elemental. For example to a program, addition is a primitive operation of the hardware. Although from the point of view of the hardware it may be complex.

pure function

A pure function always produces the same result for the same explicit inputs. Its behaviour is not dependent on some program state that can vary between calls to the function.

See Also: functional programming.

reference type

This is a kind of type in SML. Values of this type are mutable variables.

run-time

This refers to the part of the SML/NJ system that is written in C and shared by all SML/NJ programs.

SML

This is an abbreviation for Standard ML.

See Also: SML/NJ, CML.

SML/NJ

This is the New Jersey implementation of the Standard ML language.

See Also: SML.

stdin, stdout, stderr

These are the predefined UNIX file descriptors: standard input, standard output and standard error.

signature

In SML this is a collection of declarations for types and values that a structure exports to the rest of the program.

sneak path

This is what I call it when two parts of a program communicate by reading and writing a shared variable in some obscure way. An example in C would be if two functions in different files passed data through a global variable.

static

This refers to any information that is known about a program before it is run.

See Also: dynamic.

strong pointer

This is a pointer (or reference) to a data object that is taken seriously by the garbage collector. If an object has one or more strong pointers to it then is considered to be live.

See Also: weak pointer, live data.

structure

In SML this is a named collection of declarations.

See Also: module, functor.

tuple

In SML, this is a group of anonymous data values travelling together. The word is a generalisation of the sequence: couple, triple, quadruple, quintuple, sextuple etc.

weak pointer

This is a pointer (or reference) to a data object that is not taken seriously by the garbage collector. If an object only has weak pointers to it then it is no longer considered to be live.

See Also: live data, strong pointer.

yacc

This is the traditional parser generator available on Unix. The name stands for Yet Another Compiler Compiler. A compiler compiler is a mythical tool that generates a compiler for a programming language given a description of the language. They were a hot topic in computer science in the 1960s and 1970s.

See Also: lex.