Notes on Programming Standard ML of New Jersey

(version 110.0.6)

Riccardo Pucella

Department of Computer Science
Cornell University

Last revised:
January 10, 2001

Preface

The impetus behind these notes was the desire to provide a cohesive description of Standard ML of New Jersey, an interactive compiler and environment for Standard ML. The goal is to end up with a complete user guide to the system, inclduing the libraries, the tools and the extensions, as well as a tutorial on how to write “real” applications, centered around the use of the module system and the compilation manager. Other reasons include the desire to provide hands-on examples of how to use some maybe poorly understood features or features with an interface different from what one may be used to. Examples of such features include sockets, the input and output subsystem, the readers approach to text scanning and the use of continuations to handle non-local control-flow. All in all, this would be a repository for snippets of SML and SML/NJ programming lore.

These notes are not a tutorial introduction to Standard ML. There exists excellent introductory material, available both commercially or freely over the Internet. These notes complement this literature by focusing on the Standard ML of New Jersey environment. The first part of these notes does given an overview of the language, but a quick one and without highlighting some of the subtleties of the language. Better writers than I have written better introductory material and I urge you to read those first. References are given in the chapter notes of Chapter 1. I go in somewhat more details when describing the Basis Library, since some of the underlying ideas are fundamental to the overall programming philosophy. Unfortunately, that chapter is long, boring and reads more like a reference manual than a tutorial. Throughness and precision at odds with readability. With luck, enough sample code and tutorial material is interspersed to lighten the mood. In the course of the notes, I take the opportunity to describe more advanced topics typically not covered in introductory material, such as sockets programming, signals handling, continuations, concurrency. Some of these subjects are discussed in advanced programming language courses, which not every one has taken or plan to take. Some of these subjects are hardly discussed and one needs to rummage technical papers or source code.

These notes are quite obviously a work in progress. As such, any comments or suggestions will be more than welcome. This version includes chapters 1 through 7. Planned chapters for the end of spring of 2001 include:

Chap. 8 : ML-Lex: A Lexical Analyzer Generator

Chap. 9 : ML-Yacc: A Parser Generator

Chap. 10 : Input and Output

Chap. 11 : Sockets

Chap. 12 : Regular Expressions

Chap. 13 : SML/NJ Extensions

Chap. 14 : Continuations

In the long run, chapters on the foreign function interface, CML, eXene, reader-based lexing and parsing, prettyprinting and programming reactive systems are planned, as well as sample applications including a matrix package, an interactive theorem prover, a tool to generate simple language front ends, a graphical game *à la* MineSweeper, and a simulation. Suggestions on content will also be welcome.

Notation

In the text, sample code is written in *italics*. Types are written using more mathematical notation, namely tuple types are given as $int \times int$, and function types as $int \rightarrow int$.

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

These notes describe Standard ML of New Jersey (SML/NJ), an interactive compiler, programming environment and associated tools for the programming language Standard ML (SML). The compiler is being developpped in collaboration between Bell Laboratories, Princeton University and Yale University. This chapter provides an introductory overview of the language and the environment.

1.1 Standard ML

The programming language SML has its roots as a meta-language for defining proof tactics in interactive theorem provers. Over the years, the language evolved into a full-fledged programming language, with excellent features for both small-scale and large-scale programming. Several properties make SML an interesting language. Consider the following:

- SML is mostly functional. It is based on the model of evaluating expressions as opposed to the model of executing sequences of commands found in imperative languages. The ability to treat functions as first-class values allows so-called higher-order programming, a very powerful programming technique. In contrast with purely functional languages, SML allows the use of imperative constructs such as variables, assignment and sequencing of side-effecting operations.

- SML is strongly and statically typed. Each expression in the language is assigned a type describing the values it can evaluate to, and type checking at the time of compilation ensures that only compatible operations are performed. This process eliminates many of the bugs during preliminary stages of programming an application, and greatly facilitates tracking changes to the code during revisions and upgrades. SML provides the common basic types such as integers, floating points and strings, as well as compound types such as tuples, records, lists and arrays to create complex data objects. New types can be easily defined, and moreover can be made abstract, where the representation of the values cannot be seen or examined outside of a prescribed region of the code.

- SML extends this basic notion of types with parametric polymorphism. A function gets a polymorphic type when it can operate uniformly over any value of any given type. For example, reversing a list can be done uniformly for all lists, irrespectively of the type of value stored in the list.
- SML provides an easy-to-use exception mechanism for handling exceptional conditions. At any point in the code, an exception can be raised, with the effect of aborting the current operation and returning control to the last exception handler defined for that exception. Exceptions can carry arbitrary values, including functions.

These basic features of the language are complemented by advanced facilities for the management of large-scale programs. SML boasts a state-of-the-art module system, based on the notions of structures (containing the actual code), signatures (the type of structures) and functors (parametrized structures).

A concrete instance of the use of the module system is the definition of the Basis Library, a set of standard modules providing basic facilities such as input and output, mathematical operations, string and list processing, and basic operating system interfaces. The Basis Library is supported by all compliant implementations of SML.

In addition to those key features of the language, SML provides facilities that ease the programming burden. Although the language is statically typed, the programmer rarely needs to write down type annotations in the code, as most types can be inferred by the compiler. Moreover, the management of memory is automatic, with a garbage collector in charge of reclaiming memory when data is not used anymore. This eliminates most problems surrounding stray pointers in languages with explicit memory management, such as C and C++.

1.2 Standard ML of New Jersey

SML/NJ is an interactive compiler for SML. It is interactive in that the compiler sports a toplevel loop which compiles declarations and expressions entered by the user. Such entries are compiled to native machine code, and then executed. Compiled code can be exported to a file and turned into an executable. This process contrasts with most compilers for traditional languages, which are usually batch-oriented: the compiler is invoked on a set of files and produces object-code in a file. It is also in contrast with interpreters for various languages, where the expression are not compiled to native code and executed, but rather stepped through by the evaluator. SML/NJ provides a convenient blend of efficiency and flexibility.

SML/NJ provides libraries of modules beyond the Basis Library, modules implementing commonly used data structures such as hash tables, dynamic arrays and search trees, algorithms such as sorting, and packages such as regular expressions, HTML parsing and prettyprinting.

SML/NJ supplies tools for managing projects. The compilation manager CM keeps track of dependencies between the various modules making up a project and is the preferred way of managing the compilation of an application. Roughly s-peacking, an application can be defined by a file listing the various components of the application. CM provides all the benefits of the Unix tools *make* and *makedepend*, but specialized to SML and automatically tracking dependencies between modules. CM also allows for hierarchical descriptions of application components, something *make* is known to have problems with.

The tools ML-Lex and ML-Yacc are the SML/NJ versions of the popular *lex* and *yacc* tools (or *flex* and *bison*) for Unix. ML-Lex is used to generate lexical analysers from descriptions of the tokens to recognize, and ML-Yacc generates parsers from descriptions of a grammar.

Other tools exist for more specialized compiler-writing activities, such as ML-Burg, a code-generator generator. ML-RISC, not properly speaking a tool, is a backend for code generation used by SML/NJ itself. Moreover, as we shall see later in these notes, SML/NJ is itself quite suited for writing tools.

SML/NJ supports a powerful library for concurrent programming, CML, which is based on a notion of very lightweight threads and first-class synchronous operations, providing power and flexibility at very low overhead cost. EXene is a graphical interface toolkit for X-Windows implemented in CML.

1.3 History of the system

The SML/NJ project was started in 1986 by David MacQueen at Bell Laboratories and Andrew Appel at Princeton University. Initially a project to build a SML front end for research purposes, it evolved into a complete and portable programming environment for SML, with the purpose of being employed as a "language laboratory" for programming language research. In order to back claims efficiency and to motivate the implementation of useful optimizations, the decision was made to write all supporting library code in SML. The only part of the system not implemented in SML is the runtime system (written in C), in charge mostly of the memory allocation, the garbage collection and communication with the underlying operating system.

With the convergence towards satisfying the 1997 revision of SML, version 110 came out in January 1998. Various patches to the release version corrected bugs and updated libraries. At the time of writing, the current patch release version is 110.0.6. Release version 110 is the standard stable version for general use. Internal infrastructure changes and experimental features are being tested in a series of working versions not intended to be stable or generally usable. At the time of writing, the current working version is 110.29, with major changes in the intermediate representation language. Eventually, once the working versions converge to a workable and stable system, release 111 will come out incorporating the improvements.

1.4 Availability and resources

SML/NJ is freely available for many platforms, including most modern versions of Unix (Solarix, Irix) and the Microsoft Windows operating systems (95,NT,98). The MacOS port at this present time is not complete. It should be available in the next release of the system[1]. The system can be downloaded from the main SML/NJ web site:

```
http://cm.bell-labs.com/cm/cs/what/smlnj
```

The site also contains online documentation and links to related sites. The source code is freely available. SML/NJ is distributed under the following license:

> STANDARD ML OF NEW JERSEY COPYRIGHT NOTICE, LICENSE AND DISCLAIMER.
>
> Copyright (c) 1989-1997 by Lucent Technologies
>
> Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of Lucent Technologies, Bell Labs or any Lucent entity not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

[1] A MacOS port of version 0.93 was available

> Lucent disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall Lucent be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

1.5 Installation

Depending on the operating system, SML/NJ can be installed in one of three ways. For Microsoft Windows operating systems (Windows 95, Windows NT 4.0, and later), the software is available as a self-extracting software installer which can be obtained from SML/NJ's web site. In fact, the site refers to the main repository for the software, namely the ftp site

```
ftp://ftp.research.bell-labs.com/dist/smlnj/release/110/
```

The self-extracting installer is the file `110-smlnj.exe`. For Linux systems supporting the RPM package format, such as RedHat v6.0 and others, a file in `RPMS/smlnj-110.0.6-0.i386.rpm` can be found in the release directory, and easily installed using the following command (run as an administrator):

```
rpm -if smlnj-110.0.6-0.i386.rpm
```

Finally, for all other Unix systems, installation has to proceed more or less manually. This is also the way to go if one wants the source of the compiler. To install the system manually, one first needs to download the following files from the ftp site:

`110-cm.tar.Z`	Compilation Manager
`110-config.tar.Z`	Main configuration
`110-ml-burg.tar.Z`	ML-Burg tool
`110-ml-lex.tar.Z`	ML-Lex tool
`110-ml-yacc.tar.Z`	ML-Yacc tool
`110-runtime.tar.Z`	The runtime system
`110-smlnj-lib.tar.Z`	The SML/NJ library

along with *one* of the following files, the one corresponding to the system being installed (more than one file can be downloaded in case of doubt, as the system will attempt to install the right one).

`110-bin.alpha32-unix.tar.Z`	Binaries for Unix on Alpha
`110-bin.alpha32x-unix.tar.Z`	Binaries for Unix on Alpha 32x
`110-bin.hppa-unix.tar.Z`	Binaries for Unix on Hppa
`110-bin.mipseb-unix.tar.Z`	Binaries for Unix on MIPSb
`110-bin.rs6000-unix.tar.Z`	Binaries for Unix on RS6000
`110-bin.sparc-unix.tar.Z`	Binaries for Unix on Sparc
`110-bin.x86-unix.tar.Z`	Binaries for Unix on Intel x86
`110-bin.x86-win32.tar.Z`	Binaries for Windows on Intel x86

Finally, the following files are not required, but may still be useful:

`110-cml.tar.Z`	Concurrent ML
`110-eXene.tar.Z`	eXene
`110-sml-nj.tar.Z`	Source code for SML/NJ compiler
`110-smlnj-c.tar.Z`	C-Calls library

All of the appropriate files should be downloaded in a directory, say `/usr/local/smlnj`. Untar the file `110-config.tar.Z` using for example

```
zcat 110-config.tar.Z | tar -xvf -
```

This creates a subdirectory `config/`. If you want to install anything beyond the default, such as CML or eXene, edit and modify the file `config/targets` according to the instructions in the file. Once that is done, execute the following command from directory `/usr/local/smlnj/`:

```
config/install.sh
```

It all goes according to plan, you should end up with a successful installation in `/usr/local/smlnj/`. You will want to add a path to `/usr/local/smlnj/bin/` in your `PATH` environment variable.

1.6 Getting started

To start SML/NJ, type `sml` at the command shell on either Microsoft Windows or Unix systems. Under Windows, you can alternatively click the “Standard ML of New Jersey” icon in the Start Menu (under the Programs/Standard ML of New Jersey menu, assuming a typical installation). Doing this should launch the interactive compiler and produce a banner line such as:

```
Standard ML of New Jersey, Version 110.0.6, October 31, 1999
-
```

The "-" is called the (primary) prompt, and indicates that the compiler is ready to process input. The SML language is expression-based, meaning that the computational model is that of evaluating expressions. Evaluating an expression can be as simple as computing a given value (for example, the number of different ways you can pick k object from a bag containing n objects), or have complex side effects (for example, printing information to the terminal, or creating a user interface). Although expressions which correspond to whole applications are very complex, SML provides mechanisms to help manage the complexity, in the form of both data and functional abstractions.

We begin by looking at simple expressions. The simplest expression is a constant, such as *1*, *2*, *3*, or *true* or *false*, or *3.141592*. To evaluate an expression at the prompt (we also use the term "evaluate an expression at top level"), you simply enter the expression to evaluate, followed by a ";" (semicolon) and pressing RETURN. The semicolon indicates to the compiler that you have finished entering the expression and that it should start evaluating. If a semicolon is not entered, but RETURN is pressed, the compiler will present a new prompt (the secondary prompt, by default "=") and again ask for input, which will be considered a continuation of the previously entered expression. In this way, expressions spanning multiple lines can be easily entered.[2] In any case, let's evaluate some simple expressions:

```
- 1;
val it = 1 : int
- true;
val it = true : bool
- 3.141592;
val it = 3.141592 : real
```

Although it is not clear from the above interaction, SML/NJ is a compiler, albeit an interactive one. When you enter an expression to evaluate, SML/NJ compiles the expression to machine code and then executes it. It turns out that for constants, evaluation is trivial: a constant simply evaluates to itself.

Studying the above interaction, you notice that the compiler has returned more information than just the resulting value of the evaluation. Every value in SML belongs to a type[3], which is really a set of values. Thus values *1*, *2* belong to the type *int* of integers, *true* and *false* belong to the type *bool* of booleans, and *3.141592* belongs to the type *real* of floating-point numbers.

More complex expressions can be built using operations and syntactic forms. The expression *if* exp_1 *then* exp_2 *else* exp_3 conditionally evaluates exp_2 or exp_3 if exp_1 evaluates to *true* or *false* respectively. For this to make sense, the compiler needs to enforce the fact that exp_1 evaluates to either *true* or *false*, in other words

[2] To unclutter the examples, I will often not show the seconday prompt from the sample outputs in these notes.

[3] We often use the term "a value has a type" over the more accurate "belongs to".

that the expression exp_1 belongs to type *bool*. This is part of the process called type checking. Every syntactic form and operation specifies the type of the expressions they are made up from. Before evaluating, SML/NJ will check that those constraints are satisfied. For example, a conditional expression requires a boolean condition, the addition operation expects integers arguments, and so on. Thus,

```
- if true then 0 else 1;
val it = 0 : int
```

is type correct, and can be compiled and executed, while

```
- if 0 then 0 else 1;
stdIn:34.1-34.19 Error: case object and rules don't agree [literal]
  rule domain: bool
  object: int
  in expression:
    (case 0
      of true => 0
       | false => 1)
```

fails to type check and the compiler returns a compile-time type error. The error message specifies that the compiler was expecting the condition to be have boolean type, but instead was found to be an integer, producing a type mismatch.

It is important to note that type checking does not execute code. This helps explain various restrictions. For example, when type checking a conditional expression *if* exp_1 *then* exp_2 *else* exp_3, the system does not execute the code to decide what is the resulting type. Therefore, it does not know which branch is to be chosen, and must determine the type of the result based on the fact that either branch can be executed. Since an expression can only belong to one type, it must be enforced that both branches have the same type, that is that exp_2 and exp_3 both belong to the same type. Thus,

```
- if true then 0 else 1;
val it = 0 : int
```

as we saw before is type correct, since both *0* and *1* belong to type *int*, but

```
- if true then 0 else false;
stdIn:35.1-35.26 Error: types of rules don't agree [literal]
  earlier rule(s): bool -> int
  this rule: bool -> bool
  in rule:
    false => false
```

fails to type check.

Basic operations are provided for values of various types. The arguments to operations are evaluated before the operation is performed. Arithmetic operations on integers are as expected:

```
- 3+4;
val it = 7 : int
```

Note that negative number are written as $\sim$*5*, the - sign being reserved for the subtraction operation. Other operations such as $>$, $>=$, $<$, $<=$ and = take two integers and return a boolean indicating if the specified relationship holds or not:

```
- 3>4;
val it = false : bool
```

Values can be bound to identifiers, which makes it easier to refer to those values. A value is bound using the declaration *val* id = exp, where id is an identifier and exp is an arbitrary expression. Note the distinction between a declaration (which binds identifiers to expressions) and expressions (which evaluate to values).

```
- val x = 1;
val x = 1 : int
- val pi = 3.141592;
val pi = 3.141592 : real
- val y = if (3>4) then 0 else 1;
val y = 1 : int
```

Identifiers that have been bound to values can subsequently be used in expressions to stand for that value. Given the above interaction,

```
- x + y;
val it = 2 : int
- if (x=0) then pi else 1.0;
val it = 1.0 : real
```

and so on. If you bind a value to an identifier to which a value has already been bound, the new binding shadows the old binding, and the new binding will be used from this point on. As a special case of this, whenever an expression is evaluated at toplevel and not explicitely bound to some identifier, it is automatically bound to the identifier *it*. Therefore, the value of the last evaluated expression can always be accessed. For example,

```
- 3 + 4;
val it = 7 : int
- it * 2;
val it = 14 : int
```

It is possible to declare local bindings, which are bindings valid only for the evaluation of a given expression. The expression *let* $decls$ *in* exp *end* evaluates the expression exp under the bindings specified by $decls$. After exp is evaluated, the bindings are forgotten. Pay attention to the different values of the identifier *a* in the following code:

```
- val a = 10;
val a = 10 : int
- a;
val it = 10 : int
- let  val a = 30  in a + 1 end;
val it = 31 : int
- a;
val it = 10 : int
```

Multiple declarations can be used, and are processed in turn. The following example illustrates this, as well as illustrating the use of multiple input lines:

```
- let
=   val x = 1
=   val y = x + 3
= in
=   x + y
= end;
val it = 5 : int
```

To define new operations, you use function declarations, of the form *fun* id *(*$x_1 : t_1, x_2 : t_2, \ldots$*):*$t$ = exp where id, x_1 and so on are identifiers, $t_1, t_2, \ldots, t$ are types, and exp is an expression. The expression exp can refer to the argument names $x_1, x_2, \ldots$. This declaration defines a function id which when applied to argument of the given types, evaluates the expression and returns a result of type t. Calling a function is done by supplying a value to its parameters, as in *f (*$e_1, \ldots, e_n$*)*. Note that just like for operations, the argument $e_1, \ldots, e_n$ of the function are evaluated before the function itself is applied. For example, consider a simple function to double its one argument:

```
- fun double (x:int):int = x * 2;
val double = fn : int -> int
- double (10);
val it = 20 : int
```

The type of a function is reported using an arrow notation. The type of the function *double* is *int*→*int*, meaning that it expects an argument of type *int* and returns a value of type *int*.

Functions can be recursive, that is the function can call itself. For the sake of example, we can write the following recursive version of the power function, to compute x^y given x and y:

```
- fun power (x:int,y:int):int = if (y=0) then 1 else x * power (x,y-1);
val power = fn : int * int -> int
- power (3,5);
val it = 243 : int
- 3 * 3 * 3 * 3 * 3;
val it = 243 : int
```

Mutually recursive functions need to be written together, with an attaching *and*. Consider the classical example of the even and odd predicates on natural numbers:

```
- fun even (x:int):bool = if (x<=0) then true else odd (x-1)
= and odd (x:int):bool = if (x<=0) then false else even (x-1);
val even = fn : int -> bool
val odd = fn : int -> bool
- even (10);
val it = true : bool
- odd (10);
val it = false : bool
- even (9);
val it = false : bool
- odd (9);
val it = true : bool
```

Since function declarations are declarations, functions can be declared locally, inside a *let* expression.

It can become tedious to enter all declarations by hand at the top level. SML/NJ defines a primitive operation *use* that reads the declarations and expressions from a file and processes them as if entered at the prompt. Simply create a file with your favorite editor, say `foo.sml`, and type in declarations, such as

```
fun double (x:int):int = 2 * x;
fun square (x:int):int = x * x;
fun power (x:int,y:int):int = if (y=0) then 1 else x * power (x,y-1);
```

and read it in using the *use* operation.

```
- use "foo.sml";
[opening foo.sml]
val double = fn : int -> int
val square = fn : int -> int
val power = fn : int * int -> int
val it = () : unit
```

You probably will need to provide the full path to the file. Under Unix or Windows, the path can be written as `this/is/a/path`, while under Windows one can additionally use the notation `this\\is\\a\\path`[4]. The reason why you may have to specify an explicit path is because it may not be obvious what working path SML/NJ is currently using. SML/NJ provides all the needed functionality to navigate the file system, which we will cover when discussing the Basis Library (see Chapter 4). For now let us identify some useful functions, without worrying about the details. To get the current working directory, you can call the function *OS.FileSys.getDir*[5] with a *()* argument, as in:

```
- OS.FileSys.getDir ();
val it = "/home/riccardo/work/research/working/Smlnj/started" : string
```

The function *OS.FileSys.chDir* with a string argument is used to change the current working directory, as in:

```
- OS.FileSys.chDir "/home/riccardo/work/sml";
val it = () : unit
```

If the specified directory does not exist, an error is reported

```
-  OS.FileSys.chDir "foo";

uncaught exception SysErr: No such file or directory [noent]
  raised at: <chdir.c>
```

[4]The double \\ is used instead of a single \ because \ has a special meaning as a character in strings.

[5]The dot-notation indicates the use of the module system (see Chapter 3).

Finally, for other effects, it is possible to directly call the underlying shell and execute a command there. You can call the function *OS.Process.system* with a string argument describing the command to execute, and SML/NJ will attempt to execute it, returning when the command completes. For example, under Unix, to get a listing of the current directory, you can write:

```
- OS.Process.system "ls";
#paper.tex#   foo.sml   paper.dvi  paper.ps   paper.tex~
foo.ps        paper.aux paper.log  paper.tex
val it = 0 : OS.Process.status
```

Under Windows, you should pass in the command *"dir"* instead of *"ls"* to get a similar effect.

To help navigating the file system at top level, it is helpful to have shorter mnemonic abbreviations for the above functions. You can prepare a file `defs.sml` that you *use* every SML/NJ session and that contains the following declarations:

```
fun cd (s:string):unit = OS.FileSys.chDir (s);
fun pwd ():string = OS.FileSys.getDir ();
fun ls ():int = OS.Process.system "ls";
```

(again, under Windows, you should replace *"ls"* by *"dir"*) and you can from then on call *cd "foo"*, *pwd ()* and *ls ()* to respectively change the current working directory, get the current working directory and get a listing of the current directory.

Notes

The original definition of Standard ML appeared as [74] and had an accompanying commentary [73] discussing the design and proving formal theorems about the semantics of the Definition. The 1997 revision appeared as [75] and greatly simplified various aspects of the language, removing features deemed problematic.

The design and early implementations of SML/NJ are described in [7] and [8]. The SML/NJ compiler uses a continuation-passing style intermediate representation pioneered by Steele in [101], and the details (as of 1992) are described by Appel in [5]. The intermediate representation was essentially untyped, and in 1995 Shao and Appel applied and extended the work of Leroy on representation analysis [58] with ideas from Morrisett [**?**] to produce an improved compiler that used types at later stages of the compilation process [99].

As I remarked, the original ML language was developped as a meta-language for writing proof search procedures in interactive theorem provers. LCF [40] was the first such, and actually introduced ML in the first place. Nowadays, theorem provers such as HOL [41], Isabelle [86] and NuPRL [23] all use a dialect of ML as their meta-language. In fact, HOL has recently been upgraded to use Standard ML.

Other compilers and environments for SML are available. Up until recently, Harlequin Ltd sold MLWorks, a commercial SML environment. A lightweight byte-code interpreter called MoscowML is also available. A compiler called ML-Kit [14] was developped at Diku to provide a reference implementation of SML by directly translating the semantics of the Definition. The compiler is currently used as a testbed for the study of the very interesting notion of region-based memory management, which allows the system to efficiently reclaim memory without requiring the use of a garbage collector [108]. Finally, the TIL project [105] at Carnegie-Mellon University is working on a type-based compiler that carries type information down to the lowest levels of the compilation process. Subsequent work by Morrisett and others at Cornell University on Typed Assembly Language (TAL) showed how types can be pushed down to the level of the assembly language [82, 80]. Until recently, a compiler from ML to Java called MLJ was under development [**?**]. Finally, an optimizing compiler called ML-Ton is being developed by Stephen Weeks [**?**]. This compiler uses whole-program analysis to generate extremely efficient machine code. The price to pay for such efficiency is that separate compilation is not supported.

Several dialects of ML derive from the original ML aside from SML. Most proeminent is Leroy's Objective Caml (OCaml) [63], a descendant of Caml [24] developped at INRIA. OCaml provides an efficient native code compiler, and a portable byte-code interpreter. OCaml also has support for object-oriented programming, based on the work of Rémy and Vouillon [92].

Since the introduction of the polymorphic type discipline, many languages have picked up on the idea. A relevant branch of developpment is that of purely functional languages, languages which do not allow side-effects such as assignment and exceptions. Such languages are also typically based on a lazy evaluation mechanism, where parameters to functions are not evaluated until they are needed by the body of the function. In contrast, SML is strict (or eager). Lazy ML was a lazy purely functional version of ML developped by Augustsson and Johnsson in the early 80's [10, 11]. Modern lazy languages that are not based on ML but borrowing the basic polymorphic type discipline include Miranda[6] [109] and Haskell [88].

The recent working versions of SML/NJ considerably modified the internal workings of the compiler. A fully-typed intermediate language called FLINT developped at Yale by Shao and others [98] is being integrated. Similarly, the code generation backend is being reimplemented using George's ML-RISC, a powerful library [36] that uses the full power of the SML module system to factor out commonalities between code generation for various architectures.

[6]Miranda is a trademark of Research Software Limited

Part I

Standard ML

Chapter 2

The Core Language

In this chapter, we give a brief description of the core language of SML. The core language is the small-scale part of SML, in charge of expressing types and computations. Managing the name space and separating programs into independent communicating units is the province of the module system described in Chapter 3.

As we noted in the introduction, SML is a mostly functional language, based on the notion of expression evaluation. A program is really an expression to be evaluated, returning a value. The expression may be simply to compute a mathematical value such as the roots of a polynomial, or factoring an integer, or it can be large and have visible side-effects, such as printing data or displaying a user interface for interaction with the user. A program is usually not made up of a single expression, but consists of a set of function definitions that are used by a main expression, as well as type and value declarations. In contrast, a program written in a traditional imperative language is a set of procedures made up of sequences of commands, used by a main procedure that is invoked when the program is executed.

We will presently describe the syntax and operations for writing expressions, as well as the associated type information. We then introduce more complex features such as compound types and functions, as well as imperative features such as references and exceptions.

2.1 Basic types and expressions

Expressions are ways to express computations, and computations act on values. Every value has a type, such as integer or real, denoting what type of value it is. Formally, a type can be viewed as a set of values, and the type of a value simply indicates which set the value belongs to. Generalizing, every expression has a type, namely the type of the value it evaluates to. We start our description of the language by talking about the basic values and their types. Evaluation in SML is eager. When an operation is applied to arguments, the arguments are first evaluated to values, and the operation is applied to the values so obtained. Syntactic forms have their own evaluation rules. In addition, any expression can be annotated with a type, as in *exp:ty*, or *(exp):ty*, where *exp* is an expression and *ty* is a type, which the compiler can then check.

The simplest value is *()* (pronounced unit), and it has type *unit*. The only value of type *unit* is *()*. Although seemingly useless, unit-valued expressions are typically used only for their side-effect — for all intents and purposes, they do not return a value.

Other basic values are booleans. A boolean is a value *true* or *false* of type *bool*. Boolean expressions can be built using the syntactic forms *andalso* and *orelse*, which are short-circuiting: e_1 *andalso* e_2 evaluates e_1 to a value v, and if v is *false*, the whole expression evaluates to *false*; it v is *true*, e_2 is evaluated. The *orelse* form evaluates similarly. The operation *not* is also available. The syntactic form *if* e_1 *then* e_2 *else* e_3 is used to branch on the value of a boolean expression e_1: if e_1 evaluates to *true*, then e_2 is evaluated otherwise e_3 is evaluated. Both e_2 and e_3 are required to be expressions of the same type.

Integers have type *int*. Negating an integer is done by applying the $\sim$ operation to an integer expression. Thus, -5 is written *∼5*. Other common operations on integers are available: addition (+), subtraction (-) and multiplication (*), all infix operations. Division is not implemented for integers, since in general division produces a real number. An integer division operation *div* (also infix) is available and discards the decimal portion of the result of the division. One can also compare integers with the operations $=$, $<$,$<=$, $>$, $>=$, resulting in a boolean value. SML does not impose prescribed sizes for integers, but SML/NJ uses 31-bit integers that can represent integer in the range *∼1073741824* to *1073741823*. Chapter 4 gives alternatives if larger integers are desired. Users used to languages such as C providing bit-twiddling operations on integers will note that such operations are not available for SML integers.

Floating point numbers have type *real*. They can be written as *3.141592* or as *3141592e∼6*. They can also be negated by the operation $\sim$, and the standard operations such as +, -, * and / (division) are available. A special real value *NaN* exists, to represent results of computations which do not define a real number (for example, the square root of a negative number). The special value *inf* represents ∞, for example as the result of dividing a non-zero number by zero. Real numbers can be compared by $<$, $<=$, $>$ and $>=$, but cannot be compared for equality using $=$.[1]. Various operations are available for approximating equality tests in the Basis

[1]In the previous versions of the language, it was indeed possible to test reals for equality.

Library 4. A final point to mention is that reals and integers are not interchangeable: if an integer is for example added to a real, a type error occurs. No automatic coercion of either an integer to a real or a real to an integer is performed. Coercion functions are available through the Basis Library, but they must be applied by the programmer. Although seemingly an annoying restriction, this prevents many hard-to-find problems and fits in the general safety-first approach underlying the language.

Characters have type *char*, and are written *#"a"*. Operations *ord* and *chr* for converting a character to an integer (its ASCII value) or an integer to a character are provided.

Strings have type *string*, and are written in quotes, as *"this is a string"*. Top level operations on strings include *size* to return the length of a string, ^ to concatenate two strings (an infix operation), and others. Many more operations, such as subscripting to extract a character from a string and so on are available through the Basis Library.

2.2 Tuples and records

It is often useful to create values which package many values. Compound types include tuples, records and lists.

A tuple is a finite sequence of values packaged as one value. The type of a tuple indicates the type of the elements in the sequence. An example of a tuple is *(true,3)*, which is a tuple of type $bool \times int$, made up of a boolean and an integer. The order is important: *(3,true)* is a different tuple, with a different type $int \times bool$. An arbitrary number of values can be packaged in this way, leading to correspondingly long tuple types. One way to extract a value from a tuple is to use the selection operations *#1*, *#2*, ..., which extract the element at the corresponding position in the tuple. For example,

```
- #2 (1,3.0,true);
val it = 3.0 : real
```

Section 2.4 will discuss an often better alternative to access tuple elements, via pattern matching. Note that a tuple consisting of one element is equivalent to that element alone. Moreover, a tuple of no elements is just the unit value. The syntax reflects this correspondance.

The elements of a tuple are accessed by position, which is why ordering is important. An alternative way to package values is to use records, which allow the elements to be accessed by name. A record is defined as a set of fields, such as $\{x=3.0, y=1, z=true\}$ which has type $\{x{:}real, y{:}int, z{:}bool\}$. Notice how the name of the fields is part of the type. The order of the fields is not important, but their name is. Two records with different field names but the same type of elements are still different records, as their type reflects. The value of a field is accessed by using the name of the field. Accessing field *y* of a record is done by applying the field selector *#y* to the record, as in :

```
- #y {x=3.0,y=1,z=true};
val it = 1 : int
```

In Section 2.4, we will see another way of accessing fields of records through pattern matching. The similarity between tuples and records is not accidental. A tuple $(x_1,\ldots,x_n)$ can be seen as a record $\{1{=}x_1,\ldots,n{=}x_n\}$ which explains the reliance of tuples on ordering since the ordering guides the naming of the fields, and the use of *#1*,... to access tuple elements. By the above correspondance, we have that the empty record is the unit value, since the unit value is also the empty tuple.

2.3 Declarations

Until now, we have seen how to construct basic expressions to build and handle various values. In this section, we show how to associate names with values, and how to define new types.

Value declarations

The simplest way to associate a name with a value at top level is to use the value declaration syntax, such as:

```
- val x = 3;
val x = 3 : int
```

This binds or associates the value *3* with the identifier *x*. Any expression using *x* from that point on will evaluate as if using the value *3*. If a general expression is specified, it is evaluated to a value and the value is then bound to the identifier:

```
- val y = 3+3;
val y = 6 : int
```

When an existing identifier is bound again by a declaration, it shadows the previous definition. The most recent value bound to an identifier is always used.

Multiple declarations can be written as a single declaration, as in: *val x = 1 val y = 2*. Such declarations are sequential.

One often needs to declare temporary identifiers to help evaluate an expression, for example when a given subexpression occurs often. The *let* syntactic form allows one to declare local bindings used in the evaluation of some expression. For example, the expression *(3+4) * (3+4) - (3+4)* can be written more succintly as:

```
let
   val a = 3+4
in
   (a * a) - a
end
```

Multiple declarations can appear as well:

```
let
   val a = 3+4
   val b = 8*8
in
   (a*a)-b
end
```

Although type inference usually takes care of deriving the type of the bindings, as we shall see in the following sections, it is sometimes necessary to help the type inference algorithm along with type annotations. This especially occurs in the context of polymorphic values occuring in declarations (see Section 2.6). We saw earlier that we can annotate any expression with its type, as in:

```
let
   val a = 3+4 : int
in
   (a * a) - a
end
```

It is also possible to annotate the binding itself with a type, as in:

```
let
   val a : int = 3 + 4
in
   (a * a) - a
end
```

Although the difference may seem trivial, it becomes useful in the presence of function bindings (see Section 2.5).

Type declarations

Although types often do not need to be specified due to the action of type inference, there are cases when types need to be written down. For example, to resolve polymorphic declarations (see Section 2.6), or in type specifications in signatures (see Chapter 3). Since types can sometimes grow large, it is useful to be able to name a type, the same way we can name a value using *val*. One defines a type abbreviation with the declaration:

```
- type type pair_of_ints = int * int;
type pair_of_ints = int * int
```

Note that this only defines a type abbreviation. Any value of type $int \times int$ can be used where a *pair_of_ints* is expected and vice versa. Moreover, unless explicit type annotations are used, the type inference engine will usually not report types as the defined abbreviation. For example, even after the above declaration,

```
- val a = (3,3);
val a = (3,3) : int * int
```

A type annotation however can be used to force the use of the type abbreviation.

```
- val a : pair_of_ints = (3,3);
val a = (3,3) : pair_of_ints
```

We have seen the basic types provided by the language as well as compound types for packaging value. But we have not yet created new types. We presently describe two ways of doing so.

The most straightforward way of defining a new type is to use a datatype declaration. A datatype declaration defines a type with a given name and specifies data constructors, which are used to create values of the type. The simplest use of datatypes is to construct something like enumeration types, for example

```
- datatype color = Red | Blue | Yellow;
datatype color = Blue | Red | Yellow
- val elmoColor = Red;
val elmoColor = Red : color
- val groverColor = Blue;
val groverColor = Blue : color
```

This declaration defines a type *color* whose values are *Red*, *Blue* and *Yellow*. The type *color* is a new type and values for this type can only be created by using the constructors specified in the declaration. By convention, constructor names usually start with an uppercase character. Another use of datatypes is to define types which are unions of other types. For example, suppose we wanted to define a type of elements that can either be integers or reals. One can declare

```
- datatype int_or_real = I of int | R of real;
datatype int_or_real = I of int | R of real
```

Values of this type can be created by either applying the *I* constructor to an integer or the *R* constructor to a real. The only problem at this point that we have no way of using these values! In the next section, we will see how to deconstruct such values via pattern matching.

Another extremely useful way of defining new types is to use the abstract types facility. An abstract type is a type which representation is known only to a few values. Outside of these values, the type is abstract, i.e. there is no way to access the representation of values of that type. The type system ensures that the type can only be used in the allowed way. An abstract type can be defined in SML using an *abstype* declaration, which is used just like a *datatype* declaration. For example:

```
abstype abs_color = Red | Blue | Yellow
with
   val elmoColor = Red
   val groverColor = Blue
end
```

This is just like a datatype definition, except that we specify a range of declarations in which the constructors are visible. The constructors are the "representation" of the type, that is described how the values can be built (and later deconstructed via pattern matching). This representation is hidden outside of the *abstype* construct. Only the name of the type and the declared values are exported. Looking at *elmoColor* yields:

```
- elmoColor;
val it = - : abs_color
```

The value cannot be shown, since the type is abstract. Abstract types enforce an extremely high level of safety, in that access to the internals of the value of a type is carefully controlled. This abstract type construction is less useful for large-scale programs, where the module system provides a more flexible abstract type facility.

Local declaration

The last kind of declaration we will describe is that of local declarations. Local declarations are similar in spirit to *let* expressions, but at the level of declarations. They allow the declaration of values that can be used by other declarations without themselved being revelead. For example,

```
- local
     val a = 3
     val b = 10
  in
     val x = a + b
     val y = a * b
  end;
val x = 13 : int
val y = 30 : int
- x;
val it = 13 : int
- a;
stdIn:139.1 Error: unbound variable or constructor: a
```

As in the case of abstract types, the module system (through signature thinning) provides a more flexible way of handling local declarations in the case of large-scale programs.

2.4 Pattern matching

In the previous sections, we have seen various ways to construct values of different types, and ways to construct new types. We have not focused very much attention on how to take apart values of either compound types or datatypes. We have seen, for example, fields selectors for tuples and records. We now describe pattern matching, a facility for handling all case of data deconstruction, including datatypes and many others. Pattern matching is a powerful mechanism for handling structured data. The idea revolves around the idea of a pattern: a pattern is a partial specification of the form of a data element. Variables can appear in a pattern, and cause bindings to occur for the corresponding elements in the matched object. Patterns can be used in various places. Let's examine them. First, we can use pattern matching at value declaration sites. For example,

```
- val (x,y) = (4,5);
val x = 4 : int
val y = 5 : int
```

Here, *(x,y)* is a pattern with pattern variables *x* and *y*. It is matched by the tuple *(4,5)*. If one attempts to match a value with an incompatible pattern, an error is reported:

```
- val (x,y) = (3,4,5);
stdIn:43.1-43.20 Error: pattern and expression in val dec don't agree [tycon mismatch]
  pattern:    'Z * 'Y
  expression:    int * int * int
  in declaration:
    (x,y) =
      (case (3,4,5)
        of (x,y) => (x,y))
```

Patterns can also contain litterals such as integers and booleans, which are matched exactly by themselves[2]. For example,

```
- val (3,x) = (3,5);
stdIn:44.1-44.18 Warning: binding not exhaustive
          (3,x) = ...
val x = 5 : int
- val (4,x) = (3,5);
stdIn:45.1-45.18 Warning: binding not exhaustive
          (4,x) = ...

uncaught exception nonexhaustive binding failure
  raised at: stdIn:45.1-45.18
```

Patterns can be arbitrarily complicated, for example

```
- val (x,_,((3,a),b)) = ({i=10,j=20},k=(3,4,5),((3,true),false));
stdIn:1.1-45.45 Warning: binding not exhaustive
          (x,_,((3,a),b)) = ...
val x = {i=10,j=20} : i:int, j:int
val a = true : bool
val b = false : bool
```

This example shows various features of pattern matching: one can match complex values (as *x* matching $\{i{=}10,j{=}20\}$), and the use of _ as a wildcard, always matching but causing no binding. Pattern matching for records has some specific functionality. The general pattern matching declaration for records looks like

[2]In fact, only litterals for types which admit equality are allowed in patterns. See Section 2.9.

```
- val {first=x,second=y} = {first=3,second=4};
val x = 3 : int
val y = 4 : int
```

Note again that order for records is not important, so the pattern {*second*=*y*,*first*=*x*} would also match the given value. A convenient abbreviation when one wants to completely match a field (as in the above example), is to omit the binding variable, in which case the name of the field is used as a binding name:

```
- val {first,second} = {first=3,second=4};
val first = 3 : int
val second = 4 : int
```

Moreover, if one is interested only in matching part of a record, one can use the ellipsis notation to refer to the fact that other fields may be present but one does not care about them. For example,

```
- val {second=y,...} = {first=3,second=4};
val y = 4 : int
```

There are some restrictions on when such notation can be used however, due to the nature of the type checking engine. Roughly, the type checking engine needs to know the full type of the record being matched for the ellipsis notation to be used successfully.

Finally, pattern matching can be used to access elements of datatypes. Recall our simple example of the previous section

```
datatype int_or_real = I of int | R of real
```

One can use data constructors in patterns, which are matched by a value which has been constructed by the corresponding constructor. For example,

```
- val a = I 3;
val a = I 3 : int_or_real
- val (I i) = a;
stdIn:51.1-51.14 Warning: binding not exhaustive
          I i = ...
val i = 3 : int
- val (R i) = a;
stdIn:52.1-52.14 Warning: binding not exhaustive
          R i = ...

uncaught exception nonexhaustive binding failure
  raised at: stdIn:52.1-52.14
```

Note that a match failure is raised by attempting to match a value with the wrong constructor.

Pattern matching for value declarations can also be used in *let* expressions, as in

```
- let
     val (x,y) = (4,5)
  in
     x + y
  end;
val it = 9 : int
```

A special construct allows the construction of an expression that performs a case analysis based on the result of pattern matching. A *case* expression has the following form:

```
case exp
  of pattern_1 => exp_1
   | pattern_2 => exp_2
     ...
   | pattern_n => exp_n
```

The idea is straightforward: we evaluate the expression *exp*, then attempt to match it to one of the patterns, in the given order. When a successful match occurs, the variables contained in the pattern are bound to their matching values, and the corresponding expression is evaluated and gives the result of the overall *case* expression.

A common example of the case expression is to provide alternatives depending on the value of an expression. Using the fact that litterals match only themselves, one can for example write:

```
case x
  of 0 => "zero"
   | 1 => "one"
   | _ => "out of range"
```

Notice the last pattern, which is a catch-all, since it always succeeds. A more complex example is the following, which returns the first non-zero element of a 2-tuple of integers:

```
case x
  of (0,x) => x
   | (x,_) => x
```

Here, the order of the patterns is important. If the first pattern matches, then the first non-zero cannot appear in the first position. Conversely, if the first pattern does not match, its first element cannot be zero, and we return it.

Another common case is to evaluate according to the value of a datatype. Recalling our *int_or_real* datatype example, the following expression returns a string depending on the structure of a value of the type:

```
case x
  of I _ => "integer"
   | R _ => "real"
```

It may happen that the case patterns do not cover all cases, that is, there can be values that do not match any of the patterns. In that case, SML reports a warning stating at compile-time that the matching is incomplete, and at runtime, if an unmatched expression occurs, an exception is raised (see Section 2.11). It is normally a good idea to ensure that matching is always complete. On the other hand, an actual error is reported if the patterns are redundant, that is, if because of previous patterns, one of the case patterns can never be matched. The simplest such example of this is:

```
case x
  of _ => 0
   | _ => 1
```

Clearly, this expression always yields 0 no matter what *x* is bound to. The second match can never be attempted. Redundant matches may occur due to complex pattern interaction:

```
case x
  of (_,0) => 0
   | (_,x) => x
   | (0,_) => 1
```

Again, the last pattern can never be matched, because any 2-tuple of integers matches either the first or the second pattern.

It is possible in a pattern to both match a value and decompose it further at the same time. A pattern of the form

```
x as pattern
```

both matches the overall pattern and binds the matching value to *x*, and further attempts to match the value in the pattern to the “subpattern” *pattern*. For example:

```
- val (x as (y,z),w) = ((3,4),5);
val x = (3,4) : int * int
val y = 3 : int
val z = 4 : int
val w = 5 : int
```

2.5 Functions

Until now, we have looked at expressions in isolation, figuring out basic ways of combining things together. In this section, we examine a powerful way of abstracting and reusing expressions, namely functions and function application. A function is roughly speaking a parametrized expression: it is an expression that needs to be supplied arguments before being evaluated. The process of evaluating a function after specifying the arguments is called function application. In contrast to most other languages which provide a way to parametrize expressions or sequences of commands, SML allows functions to be created dynamically and passed around in data structures. These features make SML a functional, or higher-order, language; it is also known as having first-class functions.

A function is created as follows: the expression *fn (x:t) => exp* evaluates to a function expecting a value of type *t* which will be bound to *x* and used to evaluate *exp*. As a matter of practice, we mention the type of the argument explicitely in the function, although it can be omitted, and the type of the function inferred from its definition. The expression *exp* in the above is called the body of the function. Let us look at an example. Consider the function that adds 1 to its input, which is an integer. The function is simply *fn (x:int) => x + 1*, that is it expects an integer and returns it with 1 added. Evaluating this function:

```
- fn (x:int) => x + 1;
val it = fn : int -> int
```

A function is just a value, which actually cannot be represented (hence the funny way of display the result, as a value *fn*). The type of the function is also given, *int*→*int*, namely a function expecting an integer and returning an integer. How can we use such a function? We can apply it to an integer:

```
- (fn (x:int) => x + 1) 3;
val it = 4 : int
- (fn (x:int) => x + 1) 10;
val it = 11 : int
```

Clearly, this is not optimal, having to write the function to apply everytime. However, since functions are just values, we can use the value declarations of the previous section to give a name to the function:

```
- val add1 = fn (x:int) => x + 1;
val add1 = fn : int -> int
- add1 3;
val it = 4 : int
- add1 100;
val it = 101 : int
```

Function declarations are so common that a special syntax exists[3]:

```
- fun add1 (x:int):int = x + 1;
val add1 = fn : int -> int
```

There is actually a slight difference between the two forms, which affects recursion, to which we will return in Section 2.7

Functions can actually perform pattern matching on the argument. The form:

```
fun f (x:t1):t2 => (case x
                     of pat_1 => exp_1
                      | ...
                      | pat_n => exp_n)
```

[3]When using this *fun* syntax, we also specify the result type of the function, for documentation purpose. We would have done it for *fn* expressions as well, except that the syntax does not provide any convenient place to attatch this information, aside from the body itself, as in *fn (x:int) => (x+1):int*, which is not very useful if the body of the function is very large.

can be written directly as:

```
fun f pat_1 = exp_1
  | ...
  | f pat_n = exp_n
```

Notice that the name of the function must appear at every case alternative. One disadvantage of this form over the explicit *case* form is that it does not provide a way to attach type information simply. For this reason, in these notes we will always write an explicit *case* expression, except for very simple functions.

Since parameters to functions are actually patterns, this shows how to write functions with multiple arguments: instead of passing a single value, pass a tuple of values, and use pattern matching to bind the individual values in the tuple. For example, the function:

```
- fun foo (x:int,y:int,z:int):int = x + y + z;
val foo = fn : int * int * int -> int
- foo (3,4,5);
val it = 12 : int
```

Not only can tuples be used to pass multiple arguments to functions, they can be used to return multiple results. Consider the following function that returns the sum and product of two numbers:

```
- fun sum_prod (x:int,y:int):int * int = (x+y,x*y);
val sum_prod = fn : int * int -> int * int
- sum_prod (3,4);
val it = (7,12) : int * int
```

We saw that records are just like tuples, with a different way of accessing the individual elements. Records can also be used to pass arguments to functions, and to allow the values to be passed by keyword! For example, consider the following function *full_name*:

```
- fun full_name {first:string,last:string}:string = first^" "^last;
val full_name = fn : {first:string, last:string} -> string
- full_name {last="Pucella",first="Riccardo"};
val it = "Riccardo Pucella" : string
```

Passing the arguments by keyword frees the user of the function from remembering the order in which the arguments must appear (they still have to remember the keywords though). This is especially useful when dealing with functions with many arguments of the same type. Using records in that context allow a certain amount of documentation to be included in the type of the function.

Let us now focus on the notion of functions being first-class. Why can it be useful to pass functions as arguments to other functions? Assume we have 2-tuples of integers, and we would like to transform both integers in the tuple by a given function, say *add1* or *times2*. Define:

```
- fun add1 (x:int):int = x + 1;
val add1 = fn : int -> int
- fun times2 (x:int):int = x * 2;
val times2 = fn : int -> int
```

Now let's write a function to transform 2-tuple elements by the previous functions:

```
- fun tuple_add1 (x:int,y:int):int * int = (add1 (x), add1 (y));
val tuple_add1 = fn : int * int -> int * int
- fun tuple_times2 (x:int,y:int):int * int = (times2 (x), times2 (y));
val tuple_times2 = fn : int * int -> int * int
- tuple_add1 (10,15);
val it = (11,16) : int * int
- tuple_times2 (10,15);
val it = (20,30) : int * int
```

Clearly, we can do this for any function transforming integers into integers, that is any function of type *int*→*int*. One obvious generalization is to simply write a function that applies a function passed as a parameter to the components of a tuple:

```
- fun tuple_apply (f:int->int,(x,y):int * int):int * int = (f (x),f(y));
val tuple_apply = fn : (int -> int) * (int * int) -> int * int
- tuple_apply (add1,(3,4));
val it = (4,5) : int * int
- tuple_apply (times2,(10,15));
val it = (20,30) : int * int
```

Actually, in the above example, we don't actually need to name the functions to pass to *tuple_apply*. Since we can construct function values directly using *fn*, we can simply construct one "on the fly":

```
- tuple_apply (fn (x:int) => x * 3 + 2, (10,15));
val it = (32,47) : int * int
```

Being able to construct functions on the fly and pass them around does raise serious and interesting questions. The main question is always: what happens to the free variables of the function? A free variable in a function is a variable for which there is no binding in the function. For example, if we say *fn x => x + y*, then *y* is free in the function. What happens if we call a higher-order function like *tuple_apply* with such a function containing a free variable? Which value does the free variable evaluate to?

There can be two answers to this question: (1) the variable *y* evaluates to whichever binding for *y* is valid at the time when the function is applied, or (2) the variable *y* evaluates to whatever binding for *y* was valid when and where the function was defined. Languages implementing solution (1) are said to be dynamically-scoped, those implementing solution (2) are said to be statically-scoped. SML is a statically-scoped language. Consider the following example:

```
- val foo = let val y = 3 in fn (x:int) => x + y end;
val foo = fn : int -> int
- let val y = 1000 in tuple_apply (foo,(10,20)) end;
val it = (13,23) : int * int
```

When function *foo* is defined, the variable *y* is bound to *3*. When function *foo* is applied (in the body of *tuple_apply*), the variable *y* is bound to *1000*. Since SML is statically-scoped, the value for *y* is taken to be the one present at the time of definition, that is *y* evaluates to *3*.

Suppose we foresee that we will need to apply the same function *f* to many different tuples. Using the previous *tuple_apply* function, we will need to pass around this function *f* at every *tuple_apply* call. What we can do however is write a new function that will take *f* as an argument and create a new function that will apply *tuple_apply* with that *f* as an argument. In code,

```
- fun tuple_apply2 (f:int -> int):(int * int) -> (int * int) =
    fn (x:int,y:int) => tuple_apply (f,(x,y));
val tuple_apply2 = fn : (int -> int) -> int * int -> int * int
```

Notice the type of the function: it is a function that expects a function from integers to integers and returns a function from tuples of integers to tuples of integers (recall that $\rightarrow$ associates to the right, so the type is *(int →int) →((int * int) →(int * int))*. Basically, calling *tuple_apply2* with a given integer transformation function creates a specialized *tuple_apply* function that exclusively applies *f* to tuples:

```
- val tuple_apply_times2 = tuple_apply2 times2;
val tuple_apply_times2 = fn : int * int -> int * int
- tuple_apply_times2 (3,4);
val it = (6,8) : int * int
```

Of course, anonymous functions can also be passed to *tuple_apply2*:

```
- val tuple_square = tuple_apply2 (fn (x:int) => x * x);
val tuple_square = fn : int * int -> int * int
- tuple_square (7,8);
val it = (49,64) : int * int
```

This technique of specializing a function to one of its arguments is important enough to deserve its own name, currying. To curry a function of two arguments means to turn it into a function expecting the first argument and returning a new function of the remaining argument. This generalizes straightforwardly to functions with more than two arguments. Currying is supported by syntax as well. The above example for *tuple_apply2* can be written as:

```
- fun tuple_apply2 (f:int->int) (t:int*int):(int->int) = tuple_apply (f,t);
val tuple_apply2 = fn : (int -> int) -> int * int -> int * int
- val tuple_add1 = tuple_apply2 add1;
val tuple_add1 = fn : int * int -> int * int
- tuple_add1 (3,4);
val it = (4,5) : int * int
```

Consider the example of addition. Define a function *add* to add two integers (using + directly is messy because it is defined as an infix operation):

```
- fun add (x:int,y:int):int = x + y;
val add = fn : int * int -> int
```

The curried form of the function is:

```
- fun add' (x:int) (y:int):int = x + y;
val add' = fn : int -> int -> int
```

Notice the difference in the type of the two functions. The curried *add'*, when passed a value, returns a new function that is specialized to add that value:

```
- add' 3;
val it = fn : int -> int
- it 5;
val it = 8 : int
```

We did not even bother binding the result of *add' 3*, we just use the default *it* binding. Since function application associates to the left, we can combine multiple applications as:

```
- add' 10 20;
val it = 30 : int
```

since *add' 10 20* is *(add' 10) 20*, and *add' 10* evaluates to a function, which is then applied to *20*.

When it is advantageous to curry? When a function is used often with a given value for an argument, it is sometimes simpler to curry the function definition to be able to quickly create a function specialized to that argument. Moreover, a curried function can do non-trivial processing when it receives its first argument, and by partially applying the curreied function, we don't need to incur the cost at every function call. We will see examples of this later in the notes. Currying is also often used in combinator libraries to get the right compositional behavior. The efficiency of implementation of currying is an issue. For SML/NJ, function application is optimized for the case where the argument is passed in a tuple or a record. Other systems optimize the curried case (for example, OCaml).

Although already powerful, functions in SML are made even more powerful by polymorphism and recursion, which we address in the following two sections.

2.6 Polymorphism

Polymorphism is a type discipline that allows one to write functions that can act on values of multiple types in a uniform way. The simplest example of such is the identity function, which is the function that simply returns its argument. In older typed languages, writing an identity function requires you to write many identity functions, one for each type of value. Of course, the body of all those functions is the same. In SML, one can simply write:

```
- fun identity (x:'a):'a = x;
val identity = fn : 'a -> 'a
```

The *'a* is called a *type variable*, which always start with ' in SML. In effect, this says that identity is a function that expects a value of any type, but for whatever type of value it receives, it returns that same type — hence the use of the same type variable as an argument type and as a result type. One can verify that the above function works as expected:

```
- identity (4);
val it = 4 : int
- identity (true);
val it = true : bool
```

Another example of a polymorphic function is obtained by attempting to generalize the notion of currying given at the end of the last section. Recall that currying a binary function transforms it into a function of one argument returning another function of one argument. One can write a function to generically curry a binary function *f* as follows:

```
- fun curry2 (f:'a * 'b -> 'c) = fn (x:'a) => fn (y:'b) => f (x,y);
val curry2 = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

Clearly, looking at the definition of *curry2*, one sees that the function is independent of the type of the arguments to the function *f*. In effect, *curry2* nevers "looks" at the arguments eventually passed to *f*. This "independence" is reflected in the type of *curry2*: $('a \times 'b \to 'c) \to 'a \to 'b \to 'c$. That is, given a function of type $'a \times 'b \to 'c$, a binary function of arbitrary and possibly different types returning a value of another arbitrary and possibly different type, it returns a new function of type $'a \to 'b \to 'c$ as one expects. For the sake of completeness, here's how to uncurry a given curried function:

```
- fun uncurry2 (g:'a -> 'b -> 'c):('a * 'b -> 'c) =
    fn (x:'a,y:'b) => g x y;
val uncurry2 = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

We can generalize currying and uncurrying to functions of arity greater than 2:

```
- fun curry3 (f:'a * 'b * 'c -> 'd):('a -> 'b -> 'c -> 'd) =
    fn (x:'a) => fn (y:'b) => fn (z:'c) => f (x,y,z);
val curry3 = fn : ('a * 'b * 'c -> 'd) -> 'a -> 'b -> 'c -> 'd
- fun uncurry3 (f:'a -> 'b -> 'c -> 'd): ('a * 'b * 'c -> 'd) =
    fn (x:'a,y:'b,z:'c) => f x y z;
val uncurry3 = fn : ('a -> 'b -> 'c -> 'd) -> 'a * 'b * 'c -> 'd
```

A form of polymorphism also occurs in datatype definitions. It is possible to parametrize a datatype definition using type variables. Suppose we wanted to define a data structure containing either one value or two. For integers, one can simply write a definition:

```
datatype one_or_two_ints = OneInt of int | TwoInts of int * int
```

But nothing in the definition is really specific to integers, and one can parametrize the declaration as follows:

```
datatype 'a one_or_two = One of 'a | Two of 'a * 'a
```

Notice the introduction of the type variable *'a* to parametrize the declaration. Consider a function to return the number of elements in a value of that type:

```
- fun howMany (x:'a one_or_two):int =
    (case (x)
       of (One _) = 1
        | (Two _) = 2);
val howMany = fn : 'a one_or_two -> int
- howMany (One 3);
val it = 1 : int
- howMany (Two (true,false));
val it = 2 : int
```

This function is clearly polymorphic over the parameter of the *one_or_two* type. Again, this comes about from the fact that the actual values stored in the datatype are never really considered. Compare with the following function

```
- fun sum (x:int one_or_two):int =
    (case x
       of One a => a
        | Two (a,b) => a + b);
val sum = fn : int one_or_two -> int
```

This function cannot polymorphic over the parameter type. Indeed, since we add the elements of the datatype, the values stored in the datatype better be of the type compatible with +, that is integers. The type checker ensures this is the case.

SML provides a polymorphic datatype similar to *one_or_two* which turns out to be very useful for describing optional values:

```
datatype 'a option = SOME of 'a | NONE
```

where intuitively a value *SOME (v)* of type *t option* indicates the presence of a value *v* of type *t*, and *NONE* indicates no value. We will return to this type in Chapter 4.

2.7 Recursion

Just like polymorphism, recursion can appear in two contexts, for functions and for types. Let us start with recursion for functions. A function is said to be recursive if it calls itself. If many functions call each other, they are said to be mutually recursive. In functional languages, recursion is used to model loops as found in more traditional imperative languages. Consider for example the following function to sum all integers from 0 to a given number:

```
- fun sumUpTo (n:int):int =
    (case (n)
       of 0 => 0
        | n => n + sumUpTo (n-1));
val sumUpTo = fn : int -> int
```

This recursive function contains all the elements of a general recursive function: it contains a base case (summing all elements up to 0 is just 0), which does not involve recursion, and a case which involves a recursive call. To ensure termination, we must make sure that we make progress towards the base case, that is in this case, the argument in the recursive call decreases towards 0[4]. One sees that to evaluate say *sumUpTo (4)*, one roughly ends up evaluating:

```
sumUpTo (4)
4 + sumUpTo (3)
4 + 3 + sumUpTo (2)
4 + 3 + 2 + sumUpTo (1)
4 + 3 + 2 + 1 + sumUpTo (0)
4 + 3 + 2 + 1 + 0
```

which yields the expected result 10. More complicated recursive functions can of course be written. A common example is to compute the n^{th} Fibonacci number. Recall that the Fibonacci numbers make up the series:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

which occurs with surprising frequency in nature and in art. One can generate any Fibonacci number in the sequence by adding the previous two Fibonacci numbers in the sequence. This leads to the following easy function:

```
- fun fib (n:int):int =
    (case (n)
      of 0 => 0
       | 1 => 1
       | n => fib (n-1) + fib (n-2));
val fib = fn : int -> int
```

If one attempts to compute the value of, say, *fib (5)* using the above idea of evaluating things successively, one quickly encounters a problem with this definition: it is highly inefficient, since *fib* gets computed many times with the same value, every time triggering a complete computation.

There are various ways of correcting this problem, but we'll focus on a particular one for pedagogical reasons. We can turn the function into a so-called tail recursive function, also called an iterative function. A tail recursive function is a recursive function that performs the recursive call as its last evaluation step. First, let's rewrite the *sumUpTo* function as a tail recursive function. The idea is instead of adding the current value to the recursive call, we pass in an extra parameter to the function that contains the result accumulated until now. To hide this extra parameter to the external interface of the function, we define a local function:

[4]Notice that if we call *sumUpTo* with a negative integer, the call never terminates.

```
fun sumUpTo' (n:int):int = let
   fun sumUpToIter (n:int,acc:int):int =
     (case n
        of 0 => acc
         | _ => sumUpToIter (n-1,acc+n))
in
   sumUpToIter (n,0)
end
```

Let us see how *sumUpTo' (4)* evaluates:

```
sumUpTo' (4)
sumUpToIter (4,0)
sumUpToIter (3,4)
sumUpToIter (2,7)
sumUpToIter (1,9)
sumUpToIter (0,10)
10
```

Notice that we do not construct a longish expression that gets evaluated at the end as in the *sumUpTo* case. For Fibonacci numbers, we use a similar idea, but pass in two extra arguments to the iterative function, namely the last two fibonacci numbers computed.

```
fun fib' (n) = let
   fun fib_iter (0,a,b) = b
     | fib_iter (n,a,b) = fib_iter (n-1,a+b,a)
in
   fib_iter (n,1,0)
end
```

It often happens that one needs a number of functions that need to call each other. These are called mutually recursive functions. Nothing of what we have seen until now allows us to express such functions. Mutually recursive functions can be declared by using the *and* syntax. The simplest example of mutually recursive functions is the pair of functions *even* and *odd*, to determine if a given natural number is even or odd:

```
- fun even (n:int):bool =
    if (n=0) then true else odd (n-1)
  and odd (n:int):bool =
    if (n=0) then false else even (n-1);
val even = fn : int -> bool
val odd = fn : int -> bool
```

This pattern generalizes to more than two mutually recursive functions in the obvious way.

Recursion highlights a subtle difference between *fun* and *val* declarations of functions. Functions declared by *fun* are naturally recursive: *fun f (x:t):t = e* can refer to itself in *e*. If we define *val f = fn (x:t) => e*, due to the evaluation rules for value declaration, *fn (x:t) => e* is evaluated before the binding for *f* is done, and any call to *f* in *e* will be interpreted as a call to whatever binding for *f* existed before the current binding for *f* is performed. One can get a recursive function back using a *val rec* declaration: *val rec f = fn (x:t) => e*, where *f* can appear in *e* and be correctly interpreted. A *fun* declaration is in fact equivalent to a *val rec* declaration.

The other context where recursion appears is in the form of recursive datatypes. A recursive datatype is a type that refers to itself. Consider the problem of representing lists of elements, that is an arbitrary but finite number of elements of the same type. The idea is to use a linked-list representation: a list is either empty, or it contains an element followed by the rest of the list. We can express this using a parametrized recursive datatype

```
datatype 'a mylist = Empty | Element of 'a * 'a mylist
```

(we use the type name *mylist* so that this does not conflict with the primitive notion of lists in SML, presented in the next section). A sample list is built as:

```
- Element (10,Element (20,Empty));
val it = Element (10,Element (20,Empty)) : int mylist
```

Functions acting on recursive datatypes are naturally recursive. Consider the problem of counting the number of elements in a list: an empty list has no elements, otherwise it has one more element than the rest of the list. The definition is thus simply:

```
- fun length (l:'a mylist):int =
    (case l
       of Empty => 0
        | Element (x,xs) => 1 + length (xs));
val length = fn : 'a mylist -> int
```

Notice that again since *length* does not care what the type of the elements is, the result type is polymorphic in the parameter type of lists. In contrast, the function to sum the elements of a list should only work for integer lists and indeed:

```
- fun sum (l:int mylist):int =
    (case l
       of Empty => 0
        | Element (x,xs) => x + sum(xs));
val sum = fn : int mylist -> int
```

A frequent operation on lists is to transform a list into another list by transforming every element into some other element, possibly of a different type. This operation, typically called *map*, is defined by taking as arguments a function to apply to every element, and the list to transform. The function *map* is also typically written in curried form.

```
- fun map (f:'a->'b) (l:'a mylist):'b mylist =
    (case l
       of Empty => Empty
        | Element (x,xs) => Element (f (x), map f xs));
val map = fn : ('a -> 'b) -> 'a mylist -> 'b mylist
```

2.8 Lists

Lists turn out to be important enough to warrant the status of primitive type in SML. Their definition is essentially that of the *mylist* type in the previous section:

```
datatype 'a list = nil | :: of 'a * 'a list
```

where *nil* represents the empty list and *::* is actually an infix operator which is right associative. The list of the first four integers can be written *1::(2::(3::(4::nil)))* or simply *1::2::3::4::nil*. With this notation, the function *map* takes the following form

```
fun map (f:'a -> 'b) (l:'a list):'b list =
  (case (l)
    of nil => nil
     | x::xs => (f (x))::(map f xs))
```

(*map* is actually a built-in function). To simplify the use of lists, an alternative syntax is available: $[x_1,\ldots,x_n]$ is equivalent to $x_1::(\cdots::(x_n::nil))$

Although pattern matching is typically used to deconstruct a list, one can also use the explicit operations *hd* and *tl* (for head and tail), which respectively return the first element of a list and the list made up of all but the first element. Operations on lists include *length* which computes the length of a list, *rev* which reverses a list, that is constructs a new list with the elements of the original list in the reverse order, and @, an infix operation to append two lists, where $[x_1,\ldots,x_n]@[y_1,\ldots,y_m]$ evaluates to $[x_1,\ldots,x_n,y_1,\ldots,y_m]$. Various operations on lists use higher-order functions, to which we will return in Chapter 4, when we describe the list support in the Basis Library.

2.9 Equality

Equality is treated specially in SML. Equality is defined for many types (that is, one can test whether two values are the same for some definition of "same") but not all. As we noted, we cannot test reals for equality using =. What about compound types, such as tuples, records and lists? What about functions?

A type is said to admit equality if it has a well-defined equality operation. For ease of use, SML overloads the = symbol to mean equality: the definition of = is different for different types, but the same symbol is used. Among the primitive types, integers, booleans, strings and characters admit equality. Reals do not. Tuples and records admit equality if all their component types admit equality, and two tuples or two records are equal if their components are equal. Functions do not admit equality. Datatypes admit equality if every constructor's parameter admits equality, and two values are equal is they have the same constructor and their parameters are equal. Consequently, lists admit equality if the underlying element type admits equality, and are equal is they have the same length and the same elements.

What happens to type inference though? What happens if we write a polymorphic function that uses equality? A polymorphic function should work at all types, but as we've seen, equality is not defined at all types. Consider a function taking a list and a value and returning true or false depending on whether it finds the value in the list. The function is a straightforward recursive walk over the list. However, if we try to assign it the expected polymorphic type, we fail:

```
- fun find (l:'a list,y:'a):bool =
    (case l
       of nil => false
        | x::xs => (x=y) orelse find (xs,y));
stdIn:36.18-36.23 Error: operator and operand don't agree [UBOUND match]
  operator domain: ''Z * ''Z
  operand:         ''Z * 'a
  in expression:
    x = y
stdIn:36.31-36.42 Error: operator and operand don't agree [UBOUND match]
  operator domain: 'a list * 'a
  operand:         ''Z list * 'a
  in expression:
    find (xs,y)
stdIn:34.4-36.42 Error: case object and rules don't agree [UBOUND match]
  rule domain: ''Z list
  object: 'a list
  in expression:
    (case l
      of nil => false
       | x :: xs =>
           (case (x = y)
             of true => true
              | false => find <exp>))
```

This fails to type check for a simple reason. In reality, although we do not care what type of value is stored in the list, we want to ensure that whatever type is stored, equality is defined for that type. SML introduces a new kind of type variable which ranges only over types which admit equality, written *''a*. Using such a type in the function satisfies the type checker:

```
- fun find (l:''a list,y:''a):bool =
    (case l
       of nil => false
        | x::xs => (x=y) orelse find(xs,y));
val find = fn : ''a list * ''a -> bool
```

Integers admit equality, so we can use *find* on integer lists:

```
- find ([1,2,3,4,5],3);
val it = true : bool
```

Reals, on the other hand, do not admit equality. Therefore, the type checker will complain if you attempt to apply *find* on a list of reals:

```
- find ([1.0,2.0,3.0,4.0,5.0],3.0);
stdIn:87.1-87.33 Error: operator and operand don't agree [equality type required]
  operator domain: ''Z list * ''Z
  operand:         real list * real
  in expression:
    find (1.0 :: 2.0 :: <exp> :: <exp>,3.0)
```

2.10 References

All the values we have looked at until now are immutable: once the value is created, it is not possible to modify it. If we create a tuple *(3,4,5)* and wish to change its second component to say a *7*, we need to construct a new tuple. Oftentimes, it makes sense to have a variable whose value can actually change. Such variables are called *references* in SML, and have a distinct type *'a ref*, to indicate that the reference is to a value of type *'a*. A reference is created by the constructor *ref* which expects an argument, the initial content of the reference. For example,

```
- val a = ref (0);
val a = ref 0 : int ref
```

The value *a* is a reference to an integer, initially 0. Reading off the value in the reference is done through the *!* operation (called dereferencing) applied to the reference. Updating the value of a reference is done with the infix *:=* assignment operation:

```
- !a;
val it = 0 : int
- a:= 3;
val it = () : unit
- !a;
val it = 3 : int
```

Notice that the assignment operator is unit-valued. The operation is used for its side effect of changing the value in the reference.

Because of such side effects, we are justified in introducing a new syntactic form called *sequencing*. Evaluating $(e_1;\ldots;e_n)$ evaluates in sequence e_1, then e_2, and so on up to e_n, and returns the result of evaluating e_n. Clearly, since the values of e_1 up to e_{n-1} are discarded, e_1 up to e_{n-1} are only useful for their side effects. Aside from updating references, side effects also include input and output operations, which we will see in Chapter 4 and **??**.

References admit equality: two references are equal if they are the same reference, not if they contain the same value (although clearly, two equal references must hold the same value!). Thus equality for references is similar to pointer equality in languages such as C.

References are used infrequently in common SML programming. Most uses of references as temporaries can be replaced by appropriate parameter-passing. For example, suppose we wanted to count the number of times a given element appears in a list. A traditional approach consists of looping through all the elements of the list, incrementing a variable as you go, as in

```
fun count (el, list) = let
   val cnt = ref 0
   fun loop [] = ()
     | loop (x::xs) = (if (x = el) then cnt := (!cnt) +1 else ();
                       loop (xs))
in
   loop (list);
   !cnt
end
```

However, it is much simpler to simply pass the current count through the loop, as in

```
fun count2 (el, list) = let
   fun loop ([],cnt) = cnt
     | loop (x::xs,cnt) = if (x=el) then loop (xs,cnt+1) else loop (xs,cnt)
in
   loop (list,0)
end
```

2.11 Exceptions

The final feature of the core language we look at is the exception mechanism. Raising an exception is a way to abort a computation and to return to a previous point in the evaluation. Some exceptions are built into the language or the standard library, such as the exception raised when a value does not match any of the patterns of a *case* expression, or exceptions raised to signal the overflow of arithmetic operations. Exceptions can also be defined by the programmer. Exceptions are declared by

```
exception MyException of string
exception MyException2
```

An exception can optional carry a value, whose type is specified in the declaration. Every exception has type *exn*. The *exception* declaration can be seen as adding a new constructor to the extensible *exn* datatype.

At any point in the evaluation of an expression, it is possible to signal or raise an exception by using the syntactic form *raise*. In the example above, one could say *raise MyException "some error"*. The value carried by the exception (if any) must be indicated at the *raise* point.

An exception can be intercepted or handled by wrapping an exception handler around an expression. The syntactic form

```
exp handle exn-pattern_1 => exp_1
         | ...
         | exn_pattern_n => exp_n
```

is used to define exception handlers. If during the evaluation of *exp* an exception matching one of the handler patterns is raised, the evaluation of *exp* is aborted and the result of evaluating the corresponding *exp_k* is returned. Pattern matching for exceptions is just like pattern matching for datatypes.

If during the evaluation of an expression an exception is raised for which no handler has been defined, then the whole program aborts and an *uncaught exception error* is reported at top level.

The definition of SML predefines the exceptions *Match* and *Bind*. Other standard exceptions are defined by the Basis Library.

Notes

There exists excellent introductions to Standard ML, both commercially and freely available. The books by Paulson [87] and Ullman [110] have been revised to cover the 1997 revision of SML. A new book by Hansen and Rischel [43] is available. Harper's tutorial notes [44] are available from CMU, as well as Gilmore's notes [37]. Felleisen & Friedman's book [29] is unique and delightful. Older material prior to the 1997 revision of SML can still be useful, which include books by Reade [91] and Sokolowski [100]. Again, the official reference is the Definition [75].

The type inference algorithm used by SML was first reported by Milner in [71], who independently rediscovered earlier work by Morris [78] and Hindley [49]. The algorithm is based on Robinson's unification algorithm [95].

The remark concerning the power of functions comes about from the work on Church's λ-calculus [22], a system for reasoning about computations via functions. It is equivalent computationally to Turing machines, and it is the Church-Turing thesis that all computable functions can be expressed within it. Church original calculus was untyped, and typed versions are useful for studying languages such as SML. The main reference for the (untyped) λ-calculus is Barendregt's book [12].

Various topics in this chapter concerning the use of functions, such as higher-order functions and currying are described at length in any textbook on functional programming. Good introductions include the books by Abelson and Sussman [1] and by Bird and Wadler [13], which are otherwise very different in their emphasis.

Polymorphism is typically studied in the context of the polymorphic λ-calculus of Reynolds [94]. The polymorphic λ-calculus was independently developped by Girard, from a logical standpoint [39]. An interesting consequence of type inference in the presence of polymorphism is described by Koenig [57]: the type inference algorithm can report the presence of some non-terminating functions. Indeed, if the inferred result type of a function is a type variable that does not appear in the types of the arguments of the function (and if the function does not raise exceptions or apply continuations), then we know that the function never terminates no matter what the input. For example, consider the infinite loop function *fun loop () = loop ()* which has type *unit* $\rightarrow$*'a*. This characteristic of type inference is a consequence of the Reynolds parametricity theorem [**?**].

In the text, we modified the *fib* function by turning it into a tail recursive function to improve its efficiency. Another approach would have been memoization, which in effect caches previous computations so that applying a function to a known input yields the value directly, without having to recompute it. Memoization is described in functional programming textbooks, and in [52].

Tail recursion implements an iterative process in the presence of an optimization called tail-call elimination in the compiler. Tail-call optimization is a well-known optimization in the compiler literature, but is suprisingly hard to characterize formally, because most language formalizations are not low-level enough to address the issues of memory management. One approach in that direction is given in [81]. One can convert any function to a tail recursive form via a transformation into continuation-passing style, or CPS, [79, 32, 33]. CPS representations have been used as intermediate representation for compilers for functional langages; see for example [101] or [5].

Some of the earliest languages were based on lists, most notably Lisp [68, 69, 102]. Modern functional languages often use a special notation for constructing and handling list called comprehension [111], derived from the set comprehension notation used in mathematics.

The equality operation is an example of an overloaded function, that is an operation that is defined at multiple types, with a different implementation for different types. It is a form of polymorphism, sometimes called ad-hoc, different from parametric polymorphism, which requires that the same implementation work for different types. User-defined overloaded function can be incorporated in a fully-typed setting using type classes [112]. The programming language Haskell implements overloading through type classes. Adding type classes does complicate the type system and the type inference engine. For references, see [**?**].

The interaction of polymorphism and references has always been problematic. To ensure type soundness, Tofte introduced weak type variables [107] that restricted the type of polymorphic references. MacQueen generalized the idea in SML/NJ, an analysis of which can be found in [**?**]. Leroy [59] also provided a solution to the soundness of polymorphic references. In 1995, Wright showed that the full generality of polymorphic references was rarely needed, and one could get by with the simpler value polymorphism [117]. His proposal was adopted in the 1997 revision of SML.

The exception mechanism is described in [72], based on ideas from Mycroft and Monahan. Its unification with the data type mechanism is described in [3].

It has been mentionned that the *abstype* mechanism can be supplanted by an appropriate use of the module system. Indeed, the ML2000 project (in charge of designing the next generation of the ML language) will most likely abandon the use of *abstype* on such a basis. ML2000 will also introduce new features, most notably object-oriented features and greater type flexibility. Although still under developpment, the ML2000 design goals are available as [42] (as of the end of 1999). Two experimental platforms for the design of ML2000 are OCaml [63] and Moby [31].

One view of currying is that it is an indication by the programmer of how function evaluation can be staged. Staging refers to the fact that when a function receives some of its argument, some useful computation can often already occur, before the next argument is needed. The technique of partial evaluation [**?**] can be used to automatically figure out staging information.

Chapter 3

The Module System

The SML language is made up of two sublanguages: the core language, covered in the previous chapter, which is in charge of the actual code, and the module language, which is in charge of packaging elements of the core language into coherent units for modularity and reuse. Although when experimenting with the language or prototyping functions one can restrict oneself to the core language, programming applications pretty much requires the use of the module system. To emphasize this aspect, the remainder of these notes will make use of the module system in a fundamental way. SML/NJ provides a compilation manager, which we will describe in Chapter 6, to help the compilation of module-based programs, in the form of automatic dependency tracking, separate compilation and selective recompilation.

3.1 Structures

The basic element of the SML module system is the structure (which we sometimes refer to as a module). A structure is a package of possibly both types and values into a single unit. Without the possibility of declaring types in a structure, a structure would be much like a record. It turns out that allowing types dramatically changes the rules of the game.

To illustrate the discussion, let us declare a structure defining a type for stacks along with associated stack operations. Stacks are often used as introductory example since they are not completely trivial and yet are easy to understand. They are useful in that they can be seen as building blocks for other data structures, and moreover by having them as a default example, you can rather easily compare different languages basic modularity constructs.

```
structure RevStack = struct
  type 'a stack = 'a list
  exception Empty
  val empty = []
  fun isEmpty (s:'a stack):bool =
    (case s
       of [] => true
        | _ => false)
  fun top (s:'a stack):'a =
    (case s
       of [] => raise Empty
        | x::xs => x)
  fun pop (s:'a stack):'a stack =
    (case s
       of [] => raise Empty
        | x::xs => xs)
  fun push (s:'a stack,x:'a):'a stack = x::s
  fun rev (s:'a stack):'a stack = rev (s)
end
```

This code declares a structure *RevStack* of reversible functional stacks. The structure declares a type *'a stack* representing a stack of values of type *'a*, with a value *empty* representing an empty stack, and operations *isEmpty*, *top*, *pop*, *push* and *rev*. The stacks are functional: pushing a value onto a stack or popping off a value from a stack gives a new stack, leaving the original stack untouched. Imperative stacks, on the other hand, would be implemented in such a way that pushing a value onto the stack would be a destructive operation, modifying the stack in place. The stacks implemented by *RevStack* are reversible, as they allow an operation *rev* which gives a new stack containing the elements of the original stack in the reverse order. The use of such a reversing facility will become clear as this chapter progresses.

To access the elements of the *RevStack* structure, one uses the dot notation. For example, to create a new stack of integers, one can write at top level:

```
- val s : int RevStack.stack = empty;
val s = [] : int RevStack.stack
```

(Notice the explicit type annotation. The value *RevStack.empty* is an unconstrained polymorphic value, something which is not allowed at top level[1]). This creates a stack *s* of type *int RevStack.stack*. A name such as *RevStack.empty* or *RevStack.stack* is said to be fully qualified (or simply qualified).

A structure is in fact declared by a declaration of the form *structure* ⟨*name*⟩ = ⟨*struct-exp*⟩, where ⟨*struct-exp*⟩ is a structure expression. An expression *struct* ⟨*decls*⟩ *end* is the basic structure expression, and structure identifiers can also be used as structure expressions. Thus, the following declaration is legal:

```
structure AlsoRevStack = RevStack
```

[1]Creating such a value at top level does not cause an error, but rather a warning. On the other hand, the compiler instantiates the type variable to a fresh type, rendering the value all but useless.

and indeed one can check that stacks created by *AlsoRevStack.empty* can be used by *RevStack* operations and vice versa. Although seemingly not very useful, such simple structure declarations interact with signature ascription to yield a flexible mechanism to manage name visibility (see Section 3.2).

Another type of structure expression is a *let* expression, of the form *let* ⟨*struct-decl*⟩ *in* ⟨*struct-exp*⟩ *end* where ⟨*struct-decls*⟩ can contain both structure, signature and functor declarations, as well as core language declarations, and ⟨*struct-exp*⟩ is a structure expression.

A final type of structure expression, functor application, will be explored in Section 3.3.

Any type of declaration is allowed inside a structure,[2] including types, exceptions, values, functions, and even other structures. A structure within another structure is often called a substructure. Substructures can be accessed through a generalization of the dot notation. For example, to access the innermost binding of *x* in:

```
structure Foo = struct
  structure Bar = struct
    val x = 0
  end
end
```

one uses the expression *Foo.Bar.x*.

Local declarations are also available at the module system level. The general form of a *local* declaration is *local* ⟨*struct-decl*⟩ *in* ⟨*struct-decls*⟩ *end*, where again ⟨*struct-decls*⟩ can contain both module system declarations (structures, signatures and functors), as well as core language declarations.

An interesting, powerful and often abused declaration form is the *open* declaration. As the name implies, *open S* takes a structures (or a structure expression that evaluates to a structure) and opens it up, by behaving as though the declarations inside the structure *S* where declared at the point where *open* was written. In effect, this makes the bindings declared in *S* directly available. For example, if we declare the structure

```
structure A = struct
  val x = 10
  val y = 20
end
```

then opening it at top level yields:

```
- open A;
opening A
  val x : int
  val y : int
- x+y;
val it = 30 : int
```

[2]Except for signature declarations and for functor declarations, at least for SML. SML/NJ does support the declaration of functors inside structures. This extension is explored in Section **??**.

The convenience of *open* at top level is clear — it lifts every declaration to top level, so that one does not need to figure out or remember where the bindings were actually defined. But the potential for abuse and more importantly the software engineering drawbacks should be equally clear. Practically, it is very easy through a careless *open* to silently shadow an important binding, especially when the opened structure is large. From a software engineering standpoint, *open* is problematic since it comes very hard to track dependencies between different modules when some of those modules have been opened. As we will note in Chapter 6, the Compilation Manager, which automatically tracks dependencies between modules for the purpose of managing recompilation, does not handle *open* at toplevel, since *open* can by itself also introduce new structures into the top level environments (by opening a structure which contains substructures, for example).

A restrained use of *open* can be useful however. As we noted, *open* is a declaration (and indeed behaves as such), and therefore can be used wherever a declaration is permitted, including in the declarations part of *let* and *local* expressions. Returning to the structure *A* above, we can evaluate (assuming an environment where *A* has not been opened):

```
- let open A in x + y end;
val it = 30 : int
- x+y;
stdIn:19.3 Error: unbound variable or constructor: y
stdIn:19.1 Error: unbound variable or constructor: x
```

This does not directly address the problems of accidental shadowing of important identifiers or the difficulty of tracking the original source of bindings, but it does at least somewhat contain these effects to a controllable region of code.

One common use of *open* inside a structure *A* using functions from a structure *WithAVeryLongName* is as a simple shortcut: it is indeed easier and faster to simply open *WithAVeryLongName* inside *A* and call the required functions directly than it is to qualify every function call with *WithAVeryLongName*. Problems of course arise if *WithAVeryLongName* declares bindings with the same name as some bindings in *A*. A convenient way out of this dilemma avoiding the use of *open* altogether is to locally rename the structure *WithAVeryLongName*, by defining *A* along the lines of:

```
structure A = struct
  structure W = WithAVeryLongName
  ...
end
```

and simply qualify every access to names in *WithAVeryLongName* with a *W* instead of *WithAVeryLongName*. There can be no accidental shadowing of identifiers (unless of course *W* is already declared in *A*, in which case one would hopefully choose a different name than *W*) and it is always clear where the binding is originally declared, as the qualifier indicates. This trick of local renaming is used extensively throughout these notes, and is preferred over most uses of *open*.

Let us conclude this section by introducing at another running example for this chapter. Consider the problem of implementing (functional) queues, that is data structures which allow one to push values at one end and get values out at the other, in a first-in first-out order. Whereas in the case of stacks, one could keep a simple list of the values in the stack, things are not as clean for a queue. The problem is that lists are very effective at allowing one to add and remove elements from the beginning of the list, but not from the end of the list. Nevertheless, it is possible to do so. In the following implementation of queues, we enqueue an element at the end of the list, and dequeue from the head:

```
structure Queue1 = struct
  type 'a queue = 'a list
  exception Empty
  val empty = []
  fun isEmpty (q:'a queue):bool =
    (case q
       of [] => true
        | _ => false)
  fun enqueue (q:'a queue,e:'a):'a queue =
    (case q
       of [] => [e]
        | (x::xs) => x::enqueue (xs,e))
  fun head (q:'a queue):'a =
    (case q
       of [] => raise Empty
        | (x::xs) => x)
  fun dequeue (q:'a queue):unit =
    (case q
       of [] => raise Empty
        | (x::xs) => xs)
end
```

The problem with this implementation of course is one of efficiency: walking down the whole list at every insertion can be costly if the queue is long. The problem is not solved if we enqueue at the head and dequeue at the rear. It turns out that by using a pair of stacks, one can very easily implement a queue. The idea is to have a stack in which to insert the element and a stack from which to remove the elements. The only problematic case to consider is when the "out" stack is empty, yet there are still element in the queue (in the "in" stack). If a *dequeue* or a *head* call occurs at that point, we simply take the "in" stack, reverse it, and install it as the new "out" stack. Although this reversing operation can be expensive (linear in the size of "in" stack), an amortized analysis of the running time of such a queue gives every individual operation running in constant time. Here is the corresponding code (we maintain the invariant that the "out" stack is never empty, except when the whole queue is empty):

```
structure Queue2 = struct
  structure S = RevStack
  type 'a queue = ('a S.stack * 'a S.stack)
  val empty = (S.empty,S.empty)
  fun isEmpty ((inS,outS):'a queue):bool =
    S.isEmpty (inS) andalso S.isEmpty (outS)
  fun enqueue ((inS,outS):'a queue,x:'a):'a queue =
    if (S.isEmpty (outS))
      then (inS,S.push (outS,x))
    else (S.push (inS,x),outS)
  fun head ((inS,outS):'a queue):'a =
    if (S.isEmpty (outS))
      then raise Empty
    else S.top (outS)
  fun dequeue ((inS,outS):'a queue):'a queue =
    if (S.isEmpty (outS))
      then raise Empty
    else let
      val _ = S.top (outS)
      val xs = S.pop (outS)
    in
      if (S.isEmpty (xs))
        then (S.empty,S.rev (inS))
      else (inS,xs)
    end
end
```

One feels that both *Queue1* and *Queue2* have fundamentally the same interface, although in truth *Queue1* and *Queue2* give different types to *queue*, and moreover *Queue2* provides access to a substructure *S*. In the next section, we formalize this vague notion of interface and discuss the further problem that the queues in *Queue1* and *Queue2* are not abstract, that is that one can differentiate between *Queue1* and *Queue2* queues from the fact that the former uses a simple list while the latter uses a pair of stacks.

3.2 Signatures

In a sense similar to the fact that values have a type, we can ascribe to structures a so-called type, called a signature. A signature is a set of types for values in the structure, along with types and possibly signatures for substructures. Whereas structures contain declarations of the form *val x = 42*, signatures contain specifications of the form *val x : int*, and so on. A signature looks like:

```
sig
  val x : int
  val y : string
end
```

The principal signature of a structure is the signature that specifies the exact interface for the structure. Here is the principal signature for *RevStack*, which we name *REV_STACK*:

```
signature REV_STACK = sig
  type 'a stack
  exception Empty
  val empty : 'a stack
  val isEmpty : 'a stack -> bool
  val push : 'a stack * 'a -> 'a stack
  val pop : 'a stack -> 'a stack
  val top : 'a stack -> 'a
  val rev : 'a stack -> 'a stack
end
```

(by convention, signature names are all caps) Substructures are specified in signatures by giving their own signature, as in the principal signature of structure *Foo* on Page 47:

```
sig
  structure Bar : sig
    val x : int
  end
end
```

The main use of signatures is not only informative, but prescriptive: we can use explicit signatures to control the visibility of information outside a structure. This is called signature ascription. In effect, what is visible from outside a structure is all the bindings specified by the signature. By default, if no signature is given, the principal signature is inferred and used, which means everything is visible. If a signature specifying less information is given, it will be used. In order to formalize this, we introduce the concept of signature matching. A structure *A* is said to match a signature *S* if all the bindings specified by *S* are provided by *A*, with the same types (or more general types). As a sanity check, a structure always matches its principal signature. Structure *RevStack* matches the signature *REV_STACK*, but also the following signature:

```
signature STACK = sig
  type 'a stack
  exception Empty
  val empty : 'a stack
  val isEmpty : 'a stack -> bool
  val push : 'a stack * 'a -> 'a stack
  val pop : 'a stack -> 'a stack
  val top : 'a stack -> 'a
end
```

Hence a signature may specify less declarations than a structure that matches it.

Many structures may match a given signature, where every such structure can be a different implementation of the interface described by the signature. In the previous section, we saw two implementations of functional queues, both matching the following signature:

```
signature QUEUE = sig
  type 'a queue
  exception Empty
  val empty : 'a queue
  val isEmpty : 'a queue -> bool
  val enqueue : 'a queue * 'a -> 'a queue
  val head : 'a queue -> 'a
  val dequeue : 'a queue -> 'a queue
end
```

The visibility of the declarations inside a structure can be controlled by ascribing an explicit signature to a structure. For example, revisiting the definition of *RevStack* in the previous section:

```
structure Stack : sig
  exception Empty
  type 'a stack
  val empty : 'a stack
  val push : 'a stack * 'a -> 'a stack
  val pop : 'a stack -> 'a stack
  val top : 'a stack -> 'a
end = struct
  type 'a stack = 'a list
  exception Empty
  val empty = []
  fun isEmpty (s:'a stack):bool =
    (case s
       of [] => true
        | _ => false)
  fun top (s:'a stack):'a =
    (case s
       of [] => raise Empty
        | x::xs => x)
  fun pop (s:'a stack):'a stack =
    (case s
       of [] => raise Empty
        | x::xs => xs)
  fun push (s:'a stack,x:'a):'a stack = x::s
  fun rev (s:'a stack):'a stack = rev (s)
end
```

or more succintly:

```
structure Stack : STACK = struct
  type 'a stack = 'a list
  exception Empty
  val empty = []
  fun isEmpty (s:'a stack):bool =
    (case s
       of [] => true
        | _ => false)
  fun top (s:'a stack):'a =
    (case s
       of [] => raise Empty
        | x::xs => x)
  fun pop (s:'a stack):'a stack =
    (case s
       of [] => raise Empty
        | x::xs => xs)
  fun push (s:'a stack,x:'a):'a stack = x::s
  fun rev (s:'a stack):'a stack = rev (s)
end
```

or even more succintly:

```
structure Stack : STACK = RevStack
```

and it is as though the declaration of *rev* inside the structure has been forgotten! Though it can still be used through *RevStack*. The original structure *RevStack* is unaffected, and still provides all the declarations described by its own signature. Thus, *RevStack.rev* is legal, *Stack.rev* is not, and would cause a compile-time error. On the other hand, although *Stack* has different access permissions than *RevStack*, they in fact refer to the same structure! Thus, accessibility of a value in a structure depends on the signature assocaited with the name through which the structure is accessed. To showcase the fact that the structures are shared, consider the following simplified example of a structure *A* with two declarations:

```
structure A = struct
  val a = ref (0)
  val b = true
end
```

which has a principal signature:

```
sig
  val a : int ref
  val b : bool
end
```

and consider the structure obtained by "forgetting" *b*:

```
structure B : sig val a : int ref end = A
```

Our claim that *A* and *B* represent "views" of the same underlying structure can be verified by seeing how modifying the value of *A.a:=1* will be witnessed by *B.a* changing as well. In fact, this term "view" of an underlying structure is an accurate description that will be used often. In general, a new view on a structure is obtained by ascibing a signature on an existing sturcture. On the other hand, rebinding a structure to a *struct*/*end* body, even one that has been bound previously, always creates an independent structure. For example, if we define

```
structure C = struct
  val a = ref (0)
  val b = true
end
```

then *A.a* and *C.a* are independent — chaning one will not affect the other. Although again, one must be careful: the following code restaures a dependency between two seemingly disparate entities:

```
structure D = struct
  val a = A.a
  val b = true
end
```

but here again, there is a visible "link" to the jointness of part of *D* and part of *A*, namely the reference to *A* in *D*.

Up until now, we have been using a type of signature matching called transparent signature matching, although that may not have been evident in the examples we have been looking at. Succintly, transparent matching says that types defined in signatures implicitely carry their underlying representation types. Consider this simple example:

```
structure R = struct
  type hidden = int
  val a = 10
end
```

which has the following inferred principal signature

```
sig
  type hidden
  val a : int
end
```

and clearly the following is legal: *E.a+1*, since *E.a* has type *int*. However, consider the following view of *E*:

```
structure F : sig
  type hidden
  val a : hidden
end = E
```

This time, *F.a* gets type *hidden*, but *F.a+1* is still legal: although *F.a* has type *hidden*, the type-checker knows that "under the hood" the type *hidden* is the same as *int*, and the above expression type-checks. This is transparent matching: the underlying type of *hidden* is transparently seen through *hidden*. One problem with transparent matching is that the signature does not contain enough information for us to easily determine the actual types involve during type checking. One needs to refer to the implementation (i.e. the structure) to get the full picture. This flies in the face of having a signature define the interface that tells the whole picture.

What one often wants is a way to define truly abstract types at the module level, that is modules containing types for which the signature contains all the required information, and only such. This agrees well with most accounts of viewing signatures as interfaces, and greatly increases the effectiveness of signatures to help during integration of independently programmed components: requiring all the information about a module to be centrally described in a signature helps reduce the amount of hidden dependencies one has to worry about.

To implement such a view of types in signatures, SML defines an operator :> for so-called opaque signature matching, where all the information to be used in the view is taken from the signature to be matched. Consider the following variation on the previous examples:

```
structure G :> sig
  type hidden
  val a : hidden
end = E
```

Because of opaque matching (notice the use of :>), the type-checker does not see that *hidden* is implemented as *int*, and therefore *G.a+1* will fail with a type error. Indeed, the only value of type *hidden* is *G.a*, and since no operation can be performed on values of type *hidden*, we cannot do anything with it, not even print it!

It is easy to see how such an opaque matching operation supplants in functionality the *abstype* mechanism of Section 2.3. In fact, opaque signature matching can even supplant transparent signature matching. We can specify in the signature the implementation types of the types we want to make transparent. Consider:

```
structure H :> sig
  type hidden = int
  val a : hidden
end = E
```

Although opaque matching is used, we explicitely carry through the interface the underlying type of *hidden*. So *H.a+1* again type-checks. So opaque matching can subsume transparent matching, at the cost of extra annotations in the signature. We believe the modularity benefits are great enough that for the rest of these notes, we shall use opaque matching (almost) exclusively. We will return to such issues in Section 3.4.

As a final word on the subject of signature matching, consider the following slight variation of *H* above:

```
structure I :> sig
  type hidden = int
  val a : hidden
end = G
```

This fails to signature match! The problem is that *G*, which is a view of *E* with *hidden* abstract opaquely, does not match the signature given: since type *hidden* is abstract in *G*, it is not possible to "reveal" it as being an *int*. The signature contains more information than the structure (or the view) itself! It is irrelevant that the underlying structure is *E*. The view *G* is what is being used to define view *I*, and only what is accessible through view *G* is legal.

A generalization of type abbreviation is the "where type" construct, that can annotate types inside already defined signatures. Consider the signature *HIDDEN* considered previously:

```
signature HIDDEN = sig
  type hidden
  val a : hidden
end
```

we can "instantiate" the implementation of the abstract type by

```
signature HIDDEN_IMPL = HIDDEN where type hidden = int
```

And in fact, "where type" specifications can be attached to any signature expression

```
structure H' :> HIDDEN where type hidden = int  = E
```

In fact, type abbreviations in signatures is just a "sugared" form of "where type" specification. Indeed, a signature:

```
sig
  type hidden = int
  val a : hidden
end
```

is equivalent to:

```
sig
  type hidden
  val a : hidden
end where type hidden = int
```

Another useful construct is *include*, which allows a signature to be inlined into another signature:

```
signature HIDDEN2 = sig
  include HIDDEN
  val b : hidden
end
```

Finally, we mention type sharing, which becomes extremely useful when we consider functors in Section 3.3. A type sharing annotates a specification, and can be used to specify the relationship of a type to other types. The difference with *where* types is that *where* types annotate signatures, not specificartions. Consider the following signature:

```
sig
  type s
  type t
end
```

If we want to enforce that *s* and *t* always refer to the same underlying type, then we can simply add a sharing specification:

```
sig
  type s
  type t sharing type t = s
end
```

Of course, in this case, the same effect can be achieved by the signature:

```
sig
  type s
  type t = s
end
```

The fact is that type sharing turns out to be most useful when used with structure specifications. For example, if we declare the signature above with name *TEST*:

```
signature TEST = sig
  type s
  type t
end
```

then we can express that substructures *A* and *B* share the same type *s* in the following signature using a type sharing annotation:

```
sig
  structure A : TEST
  structure B : TEST sharing type B.s = A.s
end
```

In fact, sharing specifications for substructures are permitted a special form, really an abbreviation, which is often convenient. The sharing specification in the following example:

```
sig
  structure A : TEST
  structure B : TEST  sharing A = B
end
```

is just an abbreviation for:

```
sig
  structure A : TEST
  structure B : TEST sharing type B.s = A.s
                         and type B.t = A.t
end
```

There are many restrictions and subtleties in getting sharing properties to behave correctly. For example, sharing is not transitive. It is possible to specify that structure *A* and structure *B* are shared, and that structure *B* and structure *C* are shared, but not have structure *A* and structure *C* shared.

3.3 Functors

Nowhere is the use of signatures to provide interfaces to structures more useful than in the definition of functors, which are the subject of this section.

A functor is a parametrized structure: it allows the definition of a structure which depends on another structure, provided externally. Of course, we could have the dependent structure simply access an external structure directly, but that has two drawbacks: from a software engineering perspective, the drawback is that looking at the code for the dependent structure does not reveal anything about what is needed from the external structure, short of carefully analyzing the code; from a program architecture perspective, it makes the code less reusable, since we cannot easily reuse the dependent structure code with a different external structure without going in and manually change all the references to the external structure (although one could use the abbreviation approach to alleviate this problem). To make this discussion more concrete, consider the example from earlier in this chapter, about implementing queues from stacks. Our implementation of *Queue2* uses the structure *RevStack* explicitely to manage the stacks making up the queue.

Assume now that we have another implementation of reversible stacks, say *RevStack2* (we will see one such later). If we wanted to implement queues using that alternative definition of stacks, we would need to either change the code in *Queue2*, or duplicate and rename the code to use *RevStack2*. Either way, not a pleasing prospect. The real trouble is in fact more subtle: which assumptions does the *Queue2* implementation make on the functionality provided by reversible stacks? We can rather easily read off which functions and values *Queue2* expects *RevStack* to provide, but what about their types? Moreover, what if *Queue2* relied on two external structures *A* and *B*, what about the relationship between the types in *A* and *B* as used in *Queue2*?

What we need to get this to work is a single place where we can "encapsulate" all the dependency information that *Queue2* expects *RevStack* to provide. But we have already seen such a mechanism for specifying information about what a structure provides: a signature! Clearly, we can write (or derive) a signature for what functionality *Queue2* expects, as well as the types expected and *RevStack* can be used as an external implementation if it matches the signature. Moreover, any other structure matching the signature could be used in place of *RevStack*.

In order to help implement this approach to modular programming, SML provides functors, which are a convenient way of achieving this effect. A functor is declared through

```
functor FooFun (X: sig-exp) = struct-exp
```

where *sig-exp* is a signature. Instantiating such a functor, giving it a structure will bind *X* to that structure in the body of the functor, and create the appropriate structure. Instantiating a functor is akin to applying a function. Consider the following functor definition for building a queue from a reversible stack implementation:

```
functor Queue2 (S:REV_STACK) = struct
  type 'a queue = ('a S.stack * 'a S.stack)
  val empty = (S.empty,S.empty)
  fun isEmpty ((inS,outS):'a queue):bool =
    S.isEmpty (inS) andalso S.isEmpty (outS)
  fun enqueue ((inS,outS):'a queue,x:'a):'a queue =
    if (S.isEmpty (outS))
      then (inS,S.push (outS,x))
    else (S.push (inS,x),outS)
  fun head ((inS,outS):'a queue):'a =
    if (S.isEmpty (outS))
      then raise Empty
    else S.top (outS)
  fun dequeue ((inS,outS):'a queue):'a queue =
    if (S.isEmpty (outS))
      then raise Empty
    else let
      val _ = S.top (outS)
      val xs = S.pop (outS)
    in
      if (S.isEmpty (xs))
        then (S.empty,S.rev (inS))
      else (inS,xs)
    end
end
```

Instantiating the functor is a simple matter:

```
structure QueueSt = QueueStFun (RevStack)
structure QueueSt2 = QueueStFun (RevStack2)
```

Note that we have just used the functor to create queue structures based on reversible stacks, one using the original *RevStack* implementation, and one using an unspecified *RevStack2* implementation. For the unashamed purpose of getting more experience writing functors, let us write the *RevStack2* implementation. To start with, let us first write a structure *Stack2* containing an alternative implementation of stacks, to which we will add reversibility through the use of yet another functor.

The implementation of *Stack2* relies on the observation that we can represent a stack by a function which returns an element and a function that will return the next element of the stack. With this view in mind, consider the following code:

```
structure Stack2 : STACK = struct
  datatype 'a stack = St of (unit -> ('a * 'a stack))
  exception Empty
  val empty = St (fn () => raise Empty)
  fun isEmpty (s:'a stack):bool = let
    val St (f) = s
  in
    (f (); false) handle _ => true
  end
  fun push (s:'a stack,x:'a):'a stack =
    St (fn () => (x,s))
  fun top (s:'a stack):'a = let
    val St (f) = s
    val (x,_) = f ()
  in
    x
  end
  fun pop (s:'a stack):'a stack = let
    val St (f) = s
    val (_,s') = f ()
  in
    s'
  end
end
```

Recall that *STACK* was defined on page **??**. It is not quite what we need, since we need a reverse operation on functional stacks. In a very general way, we can write a functor to turn any functional stack into functional reversible stack, as follows:

```
functor RevStackFun (structure S:STACK): REV_STACK = struct
  type 'a stack = 'a S.stack
  exception Empty = S.Empty
  val empty = S.empty
  val isEmpty = S.isEmpty
  val push = S.push
  val pop = S.pop
  val top = S.top
  fun rev (s:'a stack):'a stack = let
    fun pop_all (s:'a stack):'a list = if (S.isEmpty (s))
                         then []
                       else (S.top (s))::pop_all (S.pop (s))
    fun push_all (l:'a list,s:'a stack):'a stack =
      (case l
         of [] => s
          | e::es => push_all (es,S.push (s,e)))
  in
    push_all (pop_all (s),S.empty)
  end
end
```

Finally, we can instantiate *RevStack2* by a suitable functor application:

```
structure RevStack2 = RevStackFun (structure S = Stack2)
```

Note that the signature of the functor argument is ascribed to the argument at functor instantiation, guaranteeing that the functor cannot access elements of the argument structure which are not specified in the signature. Note also that we can also specify a signature for the result of the functor:

```
functor QueueStFun (S:REV_STACK) : QUEUE =  struct
  ...
end
```

and similarly, we can specify opaque matching for the result using :> instead of *:* to specify the functor result signature.

The above declaration takes care of functors parametrized with a single argument. What if we need a structure parametrized via two or more different structures? The problem, and its solution, is akin to the similar phenomenon that occurs for functions in the Core language. Recall that we have defined functions to take single argument. We managed to pass multiple arguments to a function by passing in a single tuple of values, that is a package containing multiple values. We can apply a similar trick to "pass" multiple structures to a functor: by wrapping those structures into a single structure! Again, let us be concrete, and consider the following somewhat artificial example. Suppose we wanted to parameterize a structure through two substructures *A* and *B* considered independent. As we mentioned above, we can simply have the functor expect a structure containing those two substructures, and at the time of functor application, we can simply create the packaging structure directly:

```
functor FooFn (Arg : sig
                 structure A : A_SIG
                 structure B : B_SIG
               end) = body using Arg.A and Arg.b
```

and

```
structure Foo = FooFn (struct
                         structure A = A
                         structure B = B
                       end)
```

This pattern is so common that an appropriate abbreviation has been introduced in the Definition to handle this: we can simply forget about the surrounding structure and pass in the specification of the elements of the structure directly. Thus, the above example can be rewritten:

```
functor FooFn (structure A : A_SIG
               structure B : B_SIG) = ...

structure Foo = FooFn (structure A = A
                       structure B = B)
```

An added benefit of this notation is that we can seemingly parameterize a structure over any value or type, not just a full structure (of course, it all gets wrapped in a structure, but one forgets after a while...). For example:

```
functor BarFn (type foo
               val a : foo) = ...
```

which, again, is simply an abbreviation for the less impressive:

```
functor BarFn (Arg : sig
                 type foo
                 val a : foo
               end) = ... <using Arg>
```

It is instructive to see how the abbreviation is implemented, i.e. what it abbreviates. Returning to the above example, the Definition specifies the following rewrite (the added stuff is in italics):

```
functor FooFn (uid : struct
                 structure A : A_SIG
                 structure B : B_SIG
               end) = struct
  local
    open uid
  in
    <body>
  end
end
```

The use of *open* in the rewrite explains why we do not need to be bothered by the fact that we never specify a name for the automatically generating packaging structure. This is one of the few uses of *open* upon which we will not frown.

This dual way of defining functors is slightly disconcerting at first, if not downright confusing. If you see a functor with more than one argument, it must use the second form. If a functor has one argument, either form may be used: if the *structure* keyword appears, it uses the second form, otherwise it uses the first form. Because of this duality, for the sake of consistency, in these notes, we shall use the second form exclusively, although it is slightly more verbose in the single-argument case. On the other hand, it does provide for slightly more explicit documentation.

3.4 Programming with modules

Notes

The original version of the module system was described by MacQueen in [64], and was inspired by a design [66] for an earlier functional language, HOPE [20] (in fact, from the associated specification language CLEAR [19]). The original module system (implemented in the SML'90 version of the language) did not have opaque signature matching or type abbreviations in signatures, and included a notion of structure sharing distinct from the type sharing discussed in Section 3.2. In essence, in SML'90, every structure had a unique static identity, and structure sharing allowed one to specify not only that types in two structures were the same, but also that actual values in two structures were the same. Structure sharing was found difficult to teach and only marginally useful, so it was dropped in the SML'97 revision of the language. Type sharing, much more useful and necessary to get functorization to work properly, was kept. Dropping structure sharing turned out to greatly simplify the theory of the module system. Although the original version of the module system (as implemented in SML/NJ) did not provide opaque signature matching, it had an *abstraction* declaration (an alternative to *structure*) which played a similar role. It was less general than what is now present, since it was not possible to opaquely specify the signature of a functor. The *abstraction* declaration was not part of SML'90.

These changes have mostly been the result of extensive work on the theory of module systems, which was aimed at understanding the static semantics of modules, that is the meaning and propagation of the types. Important work include the initial work of MacQueen on dependent types [65], the work of Harper and Mitchell [48], and subsequently work by Leroy [60] and Harper and Lillibridge [47], and more recently Russo [96].

One benefit of such theoretical studies was to provide a basis for investigating extensions to the module system. For example, SML/NJ provides higher-order functors, that is functors that can be part of structure declarations and functor bodies [67]. We will examine higher-order functors in more detail in Chapter **??**. Other extensions considered in the literature include first-class modules [47, 97], recursive modules [25, 27], dynamically replaceable modules [38] and modules with object-oriented style inheritance [76].

Other important work focused on investigating the relationship between module systems and object systems. That modules and objects served fundamentally different purposes was recognized among others by Szyperski [103] and systems incorporating both modules and objects include OCaml [63] and Moby [31]. For the latter, the integration was designed so as to clearly delineate the roles of both modules and objects. For example, Moby does not provide any privacy annotation on class members, such a hiding role being releguated to the module system through signature ascription. Still to be investigated is the relationship between module systems and mixins [18], as well as component systems [104]. A preliminary design of a module system based on a notion of components can be found in [89].

Turning to different module systems, OCaml has a module system similar in essence to SML, based on the work of Leroy [60, 61]. Differences include the fact that signature matching is implicitely opaque. Transparency can be achieved through type abbreviations in signatures. Other functional language also have powerful module systems, a good example of which being the Units system of

MzScheme [34], a compiler for Scheme [56].

The roots of module-based programming abstractions can be traced back to Modula-2 [115], ideas of which survive in such languages as Modula-3 [83], Oberon [116] and Component Pascal [84]. Modules in such languages are more static and do not emphasize the role of types in the same way as SML modules do.

We saw in the previous chapter that polymorphism is studied in the abstract using the polymorphic λ-calculus. Module systems are studied using another variant of the λ-calculus, called $F_{<:}$ (pronounced F-sub) [21], a polymorphic λ-calculus with subtyping and dependent types. Subtyping is useful to reason about signature matching, while dependent types (types which may depend on the value of expressions) are useful to model the fact that structures (which are expressions) can define types. Dependent types can easily make type checking undecidable, and for various reasons a technical requirement called the phase distinction is often imposed. Roughly speaking, a module system supports a phase distinction if it is possible to reason about the type of an expression without having to reason about the value of other expressions, even in the presence of dependencies [48].

Chapter 4

The Basis Library

Like any language that attains a certain level of standardization, SML supports a standard library providing basic support for basic type conversions, list, vector and array operations, as well as input and output, date and time operations, and various levels of system calls. This library, called the Basis Library (or simply the Basis) is supported by most implementations of SML, and greatly help writing code portable across implementations.

In this chapter, we provide an overview of the Basis, and a tutorial on the uses and conventions followed by the system. It is useful to know those conventions. They allow one to quickly find a specific functionality, and if followed in one's own code, they allow the code to use the operations of the Basis directly on one's data. This will become clear when we discuss, for instance, stream readers in Section 4.3.

Throughout these notes, I will often present interfaces to structures available through various libraries. A general convention is followed to present such interfaces: when many different structures implement the same signature, I give the signature a name and present its declaration, as in:

```
signature FOO = sig ... end
```

when a single structure is available implementing a given interface, I will often directly the structure name and implemented signature, as in:

```
structure Foo : sig ... end
```

4.1 Overview

The Basis is a collection of modules predefined in the compiler's environment. Roughly half of the library is concerned with providing operations on values of various types, including integers, reals, strings, lists, vectors and arrays. The handling of substrings in strings is much refined by the introduction of a *substring* type that does not copy its elements from the underlying string. Conversion of values to and from strings is supported through a standard interface that can be used by user-defined types, and the interface interacts nicely with the input and output subsystem.

The second half of the library addresses system programming: implementation of basic input and output primitives, with support both stream and imperative implementations; system calls including handling for dates, times, processes, file system operations; and for Posix-compliant systems, a Posix interface compatible with the input and output operations elsewhere in the library. A tentative component of the library, the support for sockets, will be discussed in detail in Chapter **??**.

Some general remarks are in order before diving into the details of the provided structures, having to do with various conventions set down by the designers of the Basis, and which we will ourselves follow in the remainder of these notes. [1]. Most of the conventions have to do with naming. The name of value variables (*val* and *fun* bindings) are in mixed upper and lowercase, with a leading lower case. Type identifiers are all lower case, with underscores separating words. Signature identifiers are all upper case, with underscores separating words. Structure and functor identifiers are mixed upper and lower case, with the initial letter of words capitalized. Datatypes constructors use the convention for signature identifiers (except in a few specific cases), and exception identifiers use the convention for structures.

Various conventions apply not only to the form of names, but to the use of names. There has been an effort to consistently use the same name for similar operations in different contexts. For example, many data structures allow a *map* and an *app* operation (see Section 4.4); any data structure where such operations are defined will name these operations *map* and *app* (for example, *List.map*, *List.app*, *Vector.map*, *Vector.app*, etc.) Many structures define types with an associated comparison function: for a type *T*, the comparison function

```
val compare : T * T -> order
```

returns a value of type *order* (defined in structure *General*, but exported to top-level) defined as follows:

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

[1] Particularly, these conventions are followed by the SML/NJ Library, to which we will return in Chapter 7.

with the obvious interpretation. From this comparison function, the relational operators $>$,$>=$,$<$ and $<=$ are derived with their expected semantics. For example, $x>y$ if and only if *compare (x,y) = GREATER*, and so on. If moreover *ty* is an equality type (see Section 2.9), the operators $=$ and $<>$ can be derived as well. Types which have a clear linear ordering are provided with a *compare* function and the appropriate relational operators, whereas abstract types (for example, *OS.FileSys.file_id*) provide simply a *compare* fuction, which can be used with an implementation of ordered binary trees.

On a related note, many structures define types with various conversion functions to and from other types. When it is clear which type is being converted (when, for example, the structure declares a single type), we use the naming convention *toT* and *fromT* to name conversion functions to type *T* and from type *T*. For example, in the *WORD* signature, we find the conversion functions

```
val fromInt : Int.int -> word
val toInt : word -> Int.int
```

When a structure defines multiple types, we use the naming convention *TtoTT* and *TTfromT*, for conversions between types *T* and *TT*.

A special case of conversion function occurs for many data structures: conversion to and from strings. We will study such conversions in Section 4.3, where we introduce a generalization of string conversions based on character readers.

As a final convention, functions which perform side-effects are unit-valued, that is they return a value of type *unit*.

The structure *General*

We can now start our description of the actual content of the Basis. The first structure of interest is *General*, which provides exceptions, types and functions that are useful throughout the Basis. The signature of *General* is given in Figure 4.1. All the declarations in *General* are available at top-level. Most of the functions (*!*,*:=*, *o*) and types (*unit*,*exn*) we have seen back in Chapter 2, where they were considered primitives. We can focus here on the remaining functions. Two functions for handling exceptions are provided: *exnName* that returns a string corresponding to the name of the exception given as an argument and *exnMessage*, returning a string with an implementation-specific message corresponding to the exception given as an argument. The message contains at least the name of the exception as returned by *exnName*. The function *before* is a convenient notation for the following expression:

```
let
  val x = a
in
  b ; x
end
```

```
structure General : sig
     eqtype unit
     type exn
     exception Bind
     exception Chr
     exception Div
     exception Domain
     exception Fail of string
     exception Match
     exception Overflow
     exception Size
     exception Span
     exception Subscript
     val exnName : exn -> string
     val exnMessage : exn -> string
     datatype order = LESS | EQUAL | GREATER
     val ! : 'a ref -> 'a
     val := : ('a ref * 'a) -> unit
     val o : (('b -> 'c) * ('a -> 'b)) -> 'a -> 'c
     val before : ('a * unit) -> 'a
     val ignore : 'a -> unit
end
```

Figure 4.1: The structure *General*

which first evaluates *a* then *b*, and returns the value of *a*. Clearly, in such a context, *b* is evaluated solely for its side-effects, since its value is discarded. In a similar vein, *ignore* is a function that evaluates its argument and returns *()*. The result of the computation is discarded and thus the argument to *ignore* is evaluated simply for its side-effects. Because of the call-by-value nature of SML, the definition of ignore is simply

```
fun ignore _ = ()
```

4.2 Basic types

Having given a general overview of the Basis and described the general structures, we are ready to dive into the first half of the library, concerned with the handling of various types of values. We focus on simple values in this section, on strings in the next section, and on aggregate data structures in the next.

The Basis adds a great many operations that can handle values of basic types to the ones given in Chapter 2. In fact, the basic operations of Chapter 2 are simply top-level bindings for some of the declarations in the appropriate structures of the Basis.

```
structure Bool : sig
     datatype bool = datatype bool
     val not : bool -> bool
     val fromString : string -> bool option
     val scan : (char, 'a) StringCvt.reader -> 'a -> (bool * 'a) option
     val toString : bool -> string
end
```

Figure 4.2: The structure *Bool*

```
structure Option : sig
     datatype 'a option = NONE | SOME of 'a
     exception Option
     val getOpt : ('a option * 'a) -> 'a
     val isSome : 'a option -> bool
     val valOf : 'a option -> 'a
     val filter : ('a -> bool) -> 'a -> 'a option
     val join : 'a option option -> 'a option
     val map : ('a -> 'b) -> 'a option -> 'b option
     val mapPartial : ('a -> 'b option) -> 'a option -> 'b option
     val compose : (('a -> 'b) * ('c -> 'a option)) -> 'c -> 'b option
     val composePartial : (('a -> 'b option) * ('c -> 'a option)) -> 'c -> 'b option
end
```

Figure 4.3: The structure *Option*

Booleans

Figure 4.2 gives the signature for *Bool*, the booleans structure. Aside from the function *not*, seen in Chapter 2, it provides string conversion functions. Since these appear in most structures for basic types, we will not mention them in descriptions such as these, aside from here. The functions *fromString* and *toString* convert back and forth between boolean values and the string ”true” and ”false” (ignoring case and initial whitespace). The *scan* function is a more general form of *fromString* that can read booleans from any suitable source of characters. We return to scanning in Section 4.3.

Options

Recall from Chapter 2 that options are values of the form *SOME (v)* or *NONE*, and are used to represent values which can be present or not. The *Option* structure, whose signature is given in Figure 4.3, defines the option type and common operations that can usefully deal with option values. Note that most of these functions are easy to write directly, but are provided for convenience. The option is defined as a datatype:

```
datatype 'a option = NONE | SOME of 'a
```

The functions *getOpt*, *isSome* and *valOf* can be used to decompose option values: *getOpt (opt,a)* returns *v* if *opt* is *SOME (v)*, *a* otherwise; *isSome (opt)* returns *true* if and only if *opt* is *SOME (v)*; *valOf (opt)* returns *v* if *opt* is *SOME (v)* and raises the *Option* exception otherwise. Although *valOf* is provided, it is usually bad as a matter of style to use it to extract the value of an optional value. It is usually better to pattern-match the optional value. The reason for this is related to the convention behind the use of optional values and exceptions: exceptions should be used for truly exceptional cases, while optional values should be used when not having a result is quite reasonable. Therefore, having *valOf* raising an exception for that case is counter-conventional. All of these functions are also available at top-level (as is the type definition itself). To show that these functions are indeed easy to define, consider the definition of *getOpt*

```
fun getOpt (SOME v,_) = v
  | getOpt (NONE,a) = a
```

The remaining functions are combinators that manipulate option values. The curried function *filter f a* takes a boolean-valued function *f* and returns *SOME (a)* if *f a* is *true* and *NONE* otherwise. The function *join* eliminates multiple layers of options: *join NONE* is *NONE*, *join (SOME (v))* is *v* when *v* is itself an option value. The various map functions apply a given function to option values: *map f a* returns *NONE* if *a* is *NONE*, and *SOME (f (v))* if *a* is *SOME (v)*; *mapPartial f a* returns *NONE* if *a* is *NONE*, and returns *f (v)* (which returns an option value) if *a* is *SOME (v)*. The expression *mapPartial f* is equivalent to *join o (map f)*.

The difference between these two functions is not clear, but there are instances when one or the other is needed. Fundamentally, *map* expects a *total* function *f*, that always returns a value; *mapPartial* is satisfied with a *partial* function, that is a function that returns either a value or *NONE*, and if the function is not defined on the input (i.e. returns *NONE*), it acts as if the input was *NONE*. In effect, in *mapPartial*, there are two distinct semantical uses of options: one to describe the optionality of the argument *a*, and one to describe the partiality of the function *f*.

The final functions are more involved, but follow the above pattern. They are better understood when viewed as acting on partial functions in the sense defined above. The function *compose (f,g) a* takes a function *f* and a partial function *g*: if *g (a)* returns *NONE*, the overall call returns *NONE*; if *g (a)* returns *SOME (v)*, then *SOME (f (v))* is returned. In effect, we get *f* composed with *g*, with a result depending on whether *g* is defined for the supplied argument. The function *composePartial (f,g) a* is similar, but here both *f* and *g* are partial functions. A definition of *composePartial* is illustrative:

```
fun composePartial (f,g) a =
  (case g a
    of NONE => NONE
     | SOME v => (case f v
                    of NONE => NONE
                     | SOME v' => SOME (v')))
```

Characters

The signature *CHAR* given in Figure 4.4 is matched by at least one structure in the Basis, the *Char* structure, that provides operations on characters. Although no assumption is made in general as to the encoding of characters, the signature defines operations assuming that characters are linearly ordered and that there is a mapping from characters to integers that preserves the ordering. The *Char* structure required by the Basis specification declares a superset of the ASCII character set. The optional *WideChar* structure (also matching *CHAR*) declares a representation of characters in terms of a fixed number of bytes. [2]

Recall from Chapter 2 that characters constants are written *#"x"* for some character *x*. This value is a value of type *Char.char*. The *char* type is declared to be an equality type, that is characters can be compared for equality. Functions for comparing characters and walking the underlying linear order include: *minChar* and *maxChar*, returning the minimum and maximum characters in the ordering, *succ* and *pred*, returning the next and previous character of a given character in the ordering, as well as the relational operators $<$,$<=$, $>$, $>=$ and a *compare* function as described in Section 4.1. Functions *ord* and *chr* convert a character to an integer value and back. The function *chr* raises a *Chr* exception if the integer does not correspond to a character.

The functions *contains* and *notContains* take a string and a character as arguments and check whether the character appears (or not) in the string. Both are boolean-valued. The functions *toUpper* and *toLower* convert a character to its uppercase (respectively lowercase) form. They leave the character unchanged if the character is not a letter.

A wide class of functions in the *CHAR* signature provide tests for various classes of characters, avoiding the need to remember numeric constants. This is a good idea since numeric constants are non-portable across different character encodings. Table 4.1 presents the the available functions.

Conversion to and from strings is done using the standard *toString*, *fromString* and *scan*. Functions *toCString* and *fromCString* are also provided: the difference

[2] Release 110 of SML/NJ does not supply a *WideChar* structure.

```
signature CHAR = sig
     eqtype char
     eqtype string
     val minChar : char
     val maxChar : char
     val maxOrd : int
     val ord : char -> int
     val chr : int -> char
     val succ : char -> char
     val pred : char -> char
     val < : (char * char) -> bool
     val <= : (char * char) -> bool
     val > : (char * char) -> bool
     val >= : (char * char) -> bool
     val compare : (char * char) -> order
     val contains : string -> char -> bool
     val notContains : string -> char -> bool
     val toLower : char -> char
     val toUpper : char -> char
     val isAlpha : char -> bool
     val isAlphaNum : char -> bool
     val isAscii : char -> bool
     val isCntrl : char -> bool
     val isDigit : char -> bool
     val isGraph : char -> bool
     val isHexDigit : char -> bool
     val isLower : char -> bool
     val isPrint : char -> bool
     val isSpace : char -> bool
     val isPunct : char -> bool
     val isUpper : char -> bool
     val fromString : String.string -> char option
     val scan : (Char.char, 'a) StringCvt.reader -> 'a -> (char * 'a) option
     val toString : char -> String.string
     val fromCString : String.string -> char option
     val toCString : char -> String.string
end
```

Figure 4.4: The signature *CHAR*

Function	Checks for...
isAlpha	letter
isUpper	uppercase letter
isLower	lowercase letter
isAlphaNum	letter or decimal digit
isAscii	seven-bit ASCII character
isControl	control character (non-printable)
isDigit	0-9
isGraph	graphical (printable but not whitespace)
isHexDigit	0-9,A-F,a-f
isPrint	printable character (whitespace or visible)
isSpace	whitespace (space,newline, tab,CR,vtab,FF)
isPunct	punctuation (graphical but not alphanumeric)

Table 4.1: Character class tests

lies in how the escape sequences are converted: *toString/fromString/scan* use SML escape sequences, while *toCString/fromCString* use C escape sequences.

Strings

The discussion on characters leads us straight into a discussion of character strings. Figure 4.5 gives the signature *STRING* declaring the operations on strings. For every type of character encoding (that is for every structure matching *CHAR*), there is a corresponding structure matching *STRING*. Since *Char* is a required structure in the Basis, the corresponding structure *String* is also required. If *WideChar* is available, a structure *WideString* (strings made up of wide characters) should also be provided. The *STRING* signature specifies a substructure matching *CHAR*, which is simply the structure implementing the character encoding that the particular implementation of strings uses.

Functions in the *STRING* signature manipulate strings in many ways. The constant *maxSize* gives the size of the largest string that can be written for the current system. The function *size* (also available at top-level) gives the number of characters in the string, whereas *sub* returns the character at the given position in the string. Note that string positions (subscripts) range from 0 to the size of the string minus 1. An exception *Subscript* (from the structure *General*) is raised if the subscript is out of range. To extract substrings, one can either use *extract* or *substring*: *substring (s,a,b)* returns a new string made up of the *b* characters in *s* starting at *a*; *extract* is a bit more flexible, taking an *int option* as a last argument, so that *extract*

```
signature STRING = sig
     eqtype string
     structure Char : CHAR
     val maxSize : int
     val size : string -> int
     val sub : (string * int) -> Char.char
     val extract : (string * int * int option) -> string
     val substring : (string * int * int) -> string
     val concat : string list -> string
     val ^ : (string * string) -> string
     val str : Char.char -> string
     val implode : Char.char list -> string
     val explode : string -> Char.char list
     val map : (Char.char -> Char.char) -> string -> string
     val translate : (Char.char -> string) -> string -> string
     val tokens : (Char.char -> bool) -> string -> string list
     val fields : (Char.char -> bool) -> string -> string list
     val isPrefix : string -> string -> bool
     val compare : (string * string) -> order
     val collate : ((Char.char * Char.char) -> order) -> (string * string) -> order
     val < : (string * string) -> bool
     val <= : (string * string) -> bool
     val > : (string * string) -> bool
     val >= : (string * string) -> bool
     val fromString : String.string -> string option
     val toString : string -> String.string
     val fromCString : String.string -> string option
     val toCString : string -> String.string
end
```

Figure 4.5: The signature *STRING*

(s,a,SOME (b)) is the same as *substring (s,a,b)*, but *extract (s,a,NONE)* returns the full suffix of *s* starting at position *a*.

Not only can one extract portions of a string, but one can also concatenate strings together: the function ^ takes two strings and concatenates them together (the function is available at top-level in an infix form), while the function *concat* is more general, taking a list of strings and concatenating them together in the order in which they appear in the list. A function *str* converts a character to the string made up of only that character.

Other functions handle strings by transforming them. In many ways, the basic functions for string transformation are reducible to *explode* and *implode*, which convert a string to and from a list of the characters. One can then use the full range of list functions to process the string (see Section **??**). Simple instances of the kind of functions one can write that way warrant their own functions (avoiding the need to perform the conversion explicitely, and potentially allowing for faster implementations). The function *map* takes a function mapping characters to characters and translates every character in the string according to that function, yielding a new string. This function is equivalent to:

```
fun map f = implode o (List.map f) o explode
```

For the sake of clarity, we can express this as:

```
fun map f x = implode (List.map f (explode s))
```

More generally, *translate* takes a function mapping characters to strings, and translates every character in the input string according to the function, concatenating the results. Again, this can be expressed by:

```
fun translate f = concat o (List.map f) o explode
```

The next two functions, *tokens* and *fields*, are used to split a string into constituent substrings, given a definition of "delimiting characters" to separate the constituent substrings. A token is a non-empty maximal substring not containing any delimiting character, whereas a field is a possibly empty maximal substring not containing any delimiting characters. The string is scanned from left to right. The set of delimiting characters is represented by a predicate *char* →*bool* returning *true* for delimiting characters, *false* otherwise. In practical terms, fields can be empty, while tokens never are. For example, if we assume *#":"* as a delimiter (that is, using a predicate *fn (#":":char) => true* || _ => *false*), then the tokens of the string *"10::20:30::40"* are *["10","20","30","40"]* and the fields are *["10","","20","30","","40"]*.

Comparison functions for strings include the standard function *compare*, and the relational operators <,<=,>,>= (and since strings are equality types, =), along with a few specific comparison functions. The function *isPrefix s1 s2* returns *true* if *s1* is a prefix of *s2*, whereas *collate* derives the lexicographic order on strings based on the given ordering on characters. For example, the *compare* function can be expressed as:

```
val compare = collate Char.compare
```

since the standard ordering on strings is derived from the standard ordering on characters. If we wanted an ordering on strings that was insensitive to the case of the letters, we would first need to define a corresponding order on characters insensitive to case:

```
fun ciCharCompare (c1,c2) = let
  val c1' = Char.toLower (c1)
  val c2' = Char.toLower (c2)
in
  Char.compare (c1',c2')
end
```

This ordering is then extended to a lexicographic order on strings:

```
val ciStringCompare = String.collate ciCharCompare
```

We can verify this:

```
- ciStringCompare ("this","ThIs");
val it = EQUAL : order
- ciStringCompare ("def","ABC");
val it = GREATER : order
```

Finally, conversions functions to and from strings are provided. It may seem odd that functions to convert strings to strings exist, but the words "strings" takes on two meanings. The function *fromString* takes a string as an SML source program string, and converts it into a string with all escape sequences converted to the appropriate characters. For example, we can convert the string *ḧi*
n¨ (notice, 4 characters) and apply *fromString* to convert this string to a string containing the actual character *#*¨
n¨ (thus, a string of length 3). Conversely, *toString* takes a string and converts it into a source string, with all non-printable characters converted to the appropriate escape sequences. The function *fromCString* and *toCString* work similarly, but use C escape sequences rather than SML escape sequences (as we saw in the *CHAR* signature earlier in this section). These functions are mostly useful when dealing with tools to process source code, as we need to read our output

```
signature INTEGER = sig
     eqtype int
     val toLarge : int -> LargeInt.int
     val fromLarge : LargeInt.int -> int
     val toInt : int -> Int.int
     val fromInt : Int.int -> int
     val precision : Int.int option
     val minInt : int option
     val maxInt : int option
     val ~ : int -> int
     val * : (int * int) -> int
     val div : (int * int) -> int
     val mod : (int * int) -> int
     val quot : (int * int) -> int
     val rem : (int * int) -> int
     val + : (int * int) -> int
     val - : (int * int) -> int
     val compare : (int * int) -> order
     val > : (int * int) -> bool
     val >= : (int * int) -> bool
     val < : (int * int) -> bool
     val <= : (int * int) -> bool
     val abs : int -> int
     val min : (int * int) -> int
     val max : (int * int) -> int
     val sign : int -> Int.int
     val sameSign : (int * int) -> bool
     val fmt : StringCvt.radix -> int -> string
     val toString : int -> string
     val fromString : string -> int option
     val scan : StringCvt.radix -> (char, 'a) StringCvt.reader -> 'a -> (int * 'a) option
end
```

Figure 4.6: The signature *INTEGER*

Integers

Integers come in many flavors, all of them signaed, all of them matching the signature *INTEGER* given in Figure 4.6. The following structures are required for Basis compliance: *Int*, *FixedInt*, *LargeInt*, *Position*. The structure *Int* implements the "default" integer type, since the top-level *int* is defined to be *Int.int*. Structure *FixedInt* is the largest fixed integers, *LargeInt* the largest arbitrary precision integers, and *Position* is the type of file positions (in files and input/output streams, see Section 4.5). SML/NJ furthermore provides *Int32*, *Int31* (for which *Int* is just an abbreviation), and *Int8*. A structure *IntInf* for arbitrary precision integers is provided by SML/NJ, but as part of the SML/NJ Library (discussed in Chapter 7). It provides most of the functionality specified by the *INTEGER* signature.

The *INTEGER* signature specifies functions to convert to and from different kind of integers: *toLarge* and *fromLarge* handle conversions to and from *LargeInt.int* values, guaranteed to be the largest representable integers, whereas *toInt* and *fromInt* convert to and from the default integer type. The value of *precision* specifies a precision of *SOME v* for fixed precision integers or *NONE* for arbitrary precision integers. The value of *maxInt* and *minInt* are the representation of the largest and smallest integer, if applicable. Standard arithmetic operations ~ (negation), + (addition), - (subtraction) and * (multiplication) are provided. Integer division operations are also provided, in two flavors: a *div*/*mod* pair and a *quot*/*rem* pair. The operation *i div j* is the truncated quotient of the division of *i* by *j*, rounded towards negative infinity, while *i mod j* is the remainder of the division of *i* by *j* with the same sign as *j*. On the other hand, *quot (i,j)* is the truncated quotient of the division of *i* by *j*, but rounded towards zero, while *rem (i,j)* is the remainder of the division of *i* by *j* of the same sign as *i*. In practice *div* and *mod* are the mathematically appropriate operations, but *quot* and *rem* follow the semantics of most hardware divide instructions, and thus may be faster than their counterparts. In all integer division operations, *Overflow* is raised if the result is not representable, and *Div* is raised if the divisor is 0.

Other standard operations include *abs*, *max*, *min* for respectively the absolute value of an integer and the maximum and minimum value of a pair of integers. The function *sign* returns 1,-1 or 0 depending on whether the integer is positive, negative or equal to 0, while *sameSign* returns *true* if and only if the two integers have the same sign.

Comparison functions are the standard ones: *compare* and the relational operators (since integers are equality types, = is also defined for integers). Conversion functions to and from strings are a bit more flexible, as they allow the use of a different radix or number base for the representation. Functions *toString* and *fromString* use the decimal representation exclusively, while *scan* takes an extra argument, a radix, taken from structure *StringCvt* (see Section 4.3), that describes the number base in which the integer is to be read. Interesting radix values include *StringCvt.BIN* (for binary, base 2), *StringCvt.OCT* (for octal,base 8), *StringCvt.DEC* (for decimal, base 10) and *StringCvt.HEX* (for hexadecimal, base 16). The function *fmt* converts an integer to a string according to the given number base. Thus, *toString i* is equivalent to *fmt StringCvt.DEC i*.

Words

As we saw in Chapter 2, words are unsigned integers with the usual arithmetic operations, as well as operations acting on the underlying bit representation of the value. Words potentially give efficient access to the primitive machine word types. The signature *WORD* given in Figure 4.7 describes the provided operations. SML/NJ implements various structures matching signature *WORD*, for different sizes of words: *Word32*, *Word31*, *Word8* respectively implement words of 32, 31 and 8 bits. The structure *Word* is an abbreviation for structure *Word31*.

The arithmetic operations provided on words are similar to those in the *INTEGER* signature, with minor differences. Conversion functions to and from other word types and integers are provided in many flavors. The value *wordSize* gives the number of bits a word value (so *Word8.wordSize* is *8*, and so on). The functions *toLargeWord*, *toLargeWordX* and *fromLargeWord* convert to and from values of type *LargeWord.word*. The function *toLargeWordX* converts while performing sign extension: if *toLargeWordX* is applied to a word *w* whose leading bit (the leftmost bit in the binary representation of *w*) is *b*, so that *w=bX* for some string of binary digits *X*, then the result of the conversion will be *bb...bX*. The name "sign extension" comes from the fact that when the words are viewed as two's complement representation for integers, sign extending preserves the sign of the represented integers. The function *fromLargeWord* stores a value modulo 2^{ws} where ws is the word size. The functions *toLargeInt*, *toLargeIntX* and from *fromLargeInt* convert to and from values of type *LargeInt.int*: for *toLargeInt*, the argument is assumed to be an integer in the range $[0, 2^{ws} - 1]$ where ws is the word size (*Overflow* is raised otherwise), for *toLargeIntX*, the word is treated as a two's complement signed integer representation. The function *fromLargeInt* stores the two's complement representation of the integer. The functions *toInt*, *toIntX* and *fromInt* are similar, but convert to and from values of type *Int.int*. For *fromInt*, the integer is sign extended prior to conversion if the precision of *Int.int* is less than *wordSize*.

The arithmetic operations +,-,*,*min*,*max* are provided. The functions *div* and *mod* return the quotient and remainder of the division of their arguments, viewed as unsigned binary numbers. None of these raise *Overflow*, but *div* and *mod* raise *Div* if the divisor is equal to 0.

```
signature WORD = sig
     eqtype word
     val wordSize : int
     val toLargeWord : word -> LargeWord.word
     val toLargeWordX : word -> LargeWord.word
     val fromLargeWord : LargeWord.word -> word
     val toLargeInt : word -> LargeInt.int
     val toLargeIntX : word -> LargeInt.int
     val fromLargeInt : LargeInt.int -> word
     val toInt : word -> Int.int
     val toIntX : word -> Int.int
     val fromInt : Int.int -> word
     val orb : (word * word) -> word
     val xorb : (word * word) -> word
     val andb : (word * word) -> word
     val notb : word -> word
     val << : (word * Word.word) -> word
     val >> : (word * Word.word) -> word
     val ~>> : (word * Word.word) -> word
     val + : (word * word) -> word
     val - : (word * word) -> word
     val * : (word * word) -> word
     val div : (word * word) -> word
     val mod : (word * word) -> word
     val compare : (word * word) -> order
     val > : (word * word) -> bool
     val < : (word * word) -> bool
     val >= : (word * word) -> bool
     val <= : (word * word) -> bool
     val min : (word * word) -> word
     val max : (word * word) -> word
     val fmt : StringCvt.radix -> word -> string
     val toString : word -> string
     val fromString : string -> word option
     val scan : StringCvt.radix -> (char, 'a) StringCvt.reader -> 'a -> (word, 'a) option
end
```

Figure 4.7: The signature *WORD*

Comparison functions and string conversion functions are as for integers, including the use of number bases of type *StringCvt.radix*.

An extra set of operations available on words are logical operations acting on the bit representations. The function *orb*, *xorb*, *andb* and *notb* return respectively the bit-wise OR, the bit-wise exclusive OR (XOR), the bit-wise AND and the bit-wise complement (NOT) of their arguments. As a reminder, here are the relevant rules for those operations on bits:

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

AND	0	1
0	0	0
1	0	1

NOT	0	1
	1	0

When applied to whole words, these operations operate in a bitwise fash. The operation $<<$ is a logical shift left, filling in 0's "from the right", where each shift is a multiplication by two (modulo the word size). The operation $>>$ is a logical shift right, filling in 0's "from the left", corresponding to an integer division by two. The operation $\sim>>$ is an arithmetic shift right, where the bit to be filled in after the shift to the right is the same as the first bit of the word before the shift, thus performing sign extension in a two's complement interpretation.

Reals

Floating-point numbers (called reals in Standard ML for historical reasons) are more complicated than simple integers. The signature *REAL*, given in Figure 4.8, specifies structures implementing floating-point numbers that should follow various IEEE standards, as well as non-trapping semantics. The most notable aspect of floating-point numbers in Standard ML is that the type *real* of a floating-point number is not an equality type. That is not say that floating-point numbers do not admit comparison, it simply says that the default = operator is not overloaded on floating-point numbers. The comparison operator == specified in signature *REAL* does compare floating-point numbers for equality but has some surprising properties undesirable in a general equality operator. The main example of this would be that *Real.==(0.0,∼0.0)* is true, while *Real.==(r/0.0,r/∼0.0)* is false.

Various structures matching *REAL* can be provided. The required structure is *Real*, the default floating-point numbers representation. Optional structures *LargeReal* and *RealN* for various values of N, the number of bits in the representation of floating-point numbers, can also be provided.

Descriptions of floating-point numbers is complicated by the fact that we have special values to represent both positive and negative infinity ($+\infty$ and $-\infty$ respectively), and the special value NaN (stands for Not a Number) to represent results which are not defined: $(+\infty) + (-\infty)$, $(-\infty) - (+\infty)$, $0/0$, ∞/∞. We simply

```
signature REAL = sig
     type real
     structure Math : MATH
     val radix : int
     val precision : int
     val maxFinite : real
     val minPos : real
     val minNormalPos : real
     val posInf : real
     val negInf : real
     val + : (real * real) -> real
     val - : (real * real) -> real
     val * : (real * real) -> real
     val / : (real * real) -> real
     val *+ : real * real * real -> real
     val *- : real * real * real -> real
     val ˜ : real -> real
     val abs : real -> real
     val min : (real * real) -> real
     val max : (real * real) -> real
     val sign : real -> int
     val signBit : real -> bool
     val sameSign : (real * real) -> bool
     val copySign : (real * real) -> real
     val compare : (real * real) -> order
     val compareReal : (real * real) -> IEEEReal.real_order
     val < : (real * real) -> bool
     val <= : (real * real) -> bool
     val > : (real * real) -> bool
     val >= : (real * real) -> bool
     val == : (real * real) -> bool
     val != : (real * real) -> bool
     val ?= : (real * real) -> bool
     val unordered : (real * real) -> bool
     val isFinite : real -> bool
     val isNan : real -> bool
     val isNormal : real -> bool
     val class : real -> IEEEReal.float_class
     val fmt : StringCvt.realfmt -> real -> string
     val toString : real -> string
     val fromString : string -> real option
     val scan : (char, ’a) StringCvt.reader -> (real, ’a) StringCvt.reader
     val toManExp : real -> man : real, exp : int
     val fromManExp : man : real, exp : int -> real
     val split : real -> whole : real, frac : real
     val realMod : real -> real
     val rem : (real * real) -> real
     val nextAfter : (real * real) -> real
     val checkFloat : real ->real
     val realFloor : real -> real
     val realCeil : real -> real
     val realTrunc : real -> real
     val floor : real -> Int.int
     val ceil : real -> Int.int
     val trunc : real -> Int.int
     val round : real -> Int.int
     val toInt : IEEEReal.rounding_mode -> real -> int
     val toLargeInt : IEEEReal.rounding_mode -> real -> LargeInt.int
     val fromInt : int -> real
     val fromLargeInt : LargeInt.int -> real
     val toLarge : real -> LargeReal.real
     val fromLarge : IEEEReal.rounding_mode -> LargeReal.real -> real
     val toDecimal : real -> IEEEReal.decimal_approx
     val fromDecimal : IEEEReal.decimal_approx -> real
end
```

Figure 4.8: The signature *REAL*

outline some of the functions in this structure. More complete information can be found in the official Basis documentation. Note that unless specified otherwise, if any argument to a function is a NaN, the result will be a NaN.

Aside from the declaration of the type *real*, the *REAL* signature prescribes a substructure *Math* that implements additional mathematical operations on the type of floating-point numbers implemented by the structure. We return to the *Math* structure later in this section. . The values *maxFinite*, *minPos* and *minNormalPos* respectively represent the maximum finite number, the minimum non-zero positive number and the minimum non-zero normalized number that can be represented. The values *posInf* and *negInf* respectively represent $+\infty$ and $-\infty$.

The operations +,-,* and / are the standard operations, along with the appropriate behavior with respect to infinities. For example, we have $\infty + \infty = \infty$,$(-\infty) + (-\infty) = -\infty$, and addition or subtraction of a finite and an infinite number yields an infinite number of the appropriate sign. Similarly, multiplication via an infinite number yields an infinite number of the appropriate sign; for instance $(+\infty) \times (-\infty) = -\infty$. Finally, *0/0* is a NaN, and an infinity divided by an infinity (irrespectively of the signs) is also a NaN; dividing a finite non-zero number by zero or an infinity by a finite number produces an infinity of the appropriate sign (recall that zero's are signed). A finite number divided by an infinity is 0 with the appropriate sign. The operations *+ and *$\sim$ are a combination of the above to allow for faster implementation on some machines. The expression *+*(a,b,c)* evaluates as *a*b+c* (similarly for *$\sim$), with a behavior derivable from the above rules with respect to infinite arguments. Operations *min*, *max* and *abs* return the smaller or larger value of two numbers, and the absolute value of one. If exactly one argument to *min* or *max* is a NaN, the result is the non-Nan argument.

The function *isFinite* returns *true* if given a finite number as argument (neither a NaN nor an infinity), whereas the function *isNan* returns *true* if given a NaN as an argument.

Various functions are provided for converting floating-point numbers to integers. The functions *floor* and *ceil* on an argument *r* respectively return the largest integer not larger than *r*, and the smallest integer not less than *r*. The function *trunc* rounds its argument towards zero. The function *round* yields the integer nearest to *r* (the nearest even integer in case of a tie). If the result cannot be represented as an integer, the exception *Overflow* is raised, while *Domain* is raised on NaN arguments. The functions *realFloor*,*realCeil* and *realTrunc* are similar to their previous counterparts, buit return integer-valued floating-point numbers (for example, 1.0 or ~ 5.0). In case of infinities or NaN, they return their argument unchanged, without raising any exception.

Comparison functions come in many flavors. We focus on the standard ones. The function *unordered* returns *true* if its arguments are unordered, that is if at least one of the arguments is a NaN. The functions <,<=,> and >= return *true* if the corresponding relation holds between reals, and *false* on non-ordered arguments. The function == returns *true* if and only if neither arguments is a NaN and the arguments are equal, ignoring signs on zeroes. The function *!=* is equivalent to *not o (op ==)*. The function *compare* is as expected, given the above description of the ordering relation. It raises the exception *IEEEReal.Unordered* on unordered arguments. The Basis structure *IEEEReal*, dealing with underlying details of the IEEE implementation of floating-point numbers, will not be detailed further in these notes.

Conversion functions to and from strings are typical, including the extra function *fmt* as in the case of integers and words. The function *scan* reads a floating-point number from a character source, while *fmt* converts a floating-point number to a string, given a specification of how to display the number. Specifications are of type *StringCvt.real_fmt* and include:

SCI arg	Scientific noation *[∼]dd.dddE[∼]ddd* where *arg* is the number of digits to appear after the decimal point (defaults to 6 is *arg* is *NONE*).
FIX arg	Fixed-point notation *[∼]ddd.ddd* where *arg* specifies the number of digits to appear after the decimal point (defaults to 6 if *arg* is *NONE*).
GEN arg	Adaptive notation: either scientific or fixed-point notation, depending on the value converted, where *arg* specifies the maximum number of significant digits used (default to 12 if *arg* is *NONE*).
EXACT	Exact decimal notation: all digits are provided, infinities are printed as *inf* or *∼inf*.

In all cases, NaN values are returned as *nan* (for *EXACT*, the form is slightly different). By default, *fromString* is equivalent to *StringCvt.scanString scan* and *toString* is equivalent to *fmt (StringCvt.GEN NONE)*.

The *Math* substructure of the *REAL* signature specifies additional mathematical operations. For the default *Real* structure, the *Real.Math* substructure is also available at top-level, as simply *Math*. The signature *MATH* is given in Figure 4.9. As with other floating-point functions, functions in *Real.Math* return NaN for NaN arguments, unless specified otherwise.

The values *pi* and *e* represent the constant π (3.141592653...) and the base of the natural logarithm (2.71828182846...). The remainder of the functions are typical: *sqrt* returns the square root of its argument, *sin*, *cos* and *tan* compute the corresponding trigonometric functions given their arguments in radians. Radians

```
signature MATH = sig
     type real
     val pi : real
     val e : real
     val sqrt : real -> real
     val sin : real -> real
     val cos : real -> real
     val tan : real -> real
     val asin : real -> real
     val acos : real -> real
     val atan : real -> real
     val atan2 : (real * real) -> real
     val exp : real -> real
     val pow : (real * real) -> real
     val ln : real -> real
     val log10 : real -> real
     val sinh : real -> real
     val cosh : real -> real
     val tanh : real -> real
end
```

Figure 4.9: The signature *MATH*

are the standard way of measuring angles, but many people are still used to degrees. Converting from degrees to radians and back is dead easy:

```
fun degToRad (d) = (360.0 * r) / (2.0 * Math.pi)
fun radToDeg (r) = (2.0 * Math.pi * d) / 360.0
```

The functions *asin* and *acos* compute the arcsine and arccosine, the inverses of sine and cosine, with results normalized to their standard range, namely $[-\pi/2, \pi/2]$ for arcsine and $[0, \pi]$ for arccosine. The functions *atan* and *atan2* both compute the arctangent, the inverse of the tangent: *atan* expects a single argument, while *atan2* expects two arguments *(y,x)* and computes *atan (y/x)* with appropriate behavior when *x* is 0. The idea behind *atan2* is that the two arguments represent a point *(y,x)* in the plane, and *atan2* computes the angle of the positive x-axis with the point, in the interval $[-\pi, \pi]$.

The exponential function *exp (x)* and *pow (x,y)* return e^x and x^y respectively, while *ln* and *log10* return the natural logarithm and the logarithm in base 10 respectively of their argument. Computing the logarithm of *a* in an arbitrary base *b* can be computed through

```
fun log (a,b) = (Math.ln (a))/(Math.ln (b))
```

Finally, *sinh*, *cosh* and *tanh* compute the corresponding hyperbolic functions.

For all those functions, a value of NaN is returned if the function is not defined for its argument, for example if a negative number is passed to *sqrt*, or if a number with a magnitude more than 1.0 is passed to an inverse trigonometric function. Infinites are returned in various cases (for example, periodically for the *tan* function), and conversely some functions are well-defined for infinite arguments (for example, *tanh (posInf)* evaluates to $\pi/2$).

4.3 More on strings

In this section, we describe two aspects of strings that are both useful and under-explained in the existing literature, namely the issue of efficiently handling substrings, and that of managing conversions from strings via the use of character sources and readers.

Substrings

Handling of substrings is provided by a structure *Substring* with a signature given in Figure 4.10. In fact, a structure matching *SUBSTRING* should be provided for every structure matching *STRING*. The substructure declaration *String* in *SUBSTRING* denotes the structure of the underlying type of strings (*Substring.String* is equivalent to *String*, *WideSubstring.String* is equivalent to *WideString*, etc). For definiteness, we discuss the *Substring* structure, which corresponds to the *String* structure.

The type *Substring.substring* can be understood as a triple *(s,i,n)* with *s* a string, *i* the starting position of the substring in *s* and *n* the length of the substring. Of course, implementations are free to implement substrings as they wish. This representation helps explain why the use of substrings (as opposed to manipulating substrings through normal string operations like *String.extract* and so on) ought to be more efficient: no new string is created (and thus no copying performed) until it is absolutely necessary (i.e. when converting a substring into a string).

The basic functions for substrings handle the extraction of substrings from strings, the conversion to basic strings and provide basic information. Extracting a substring is the job of *extract*, which takes a string *s*, a starting position *i* and an option value: if *SOME (l)*, the substring in *s* starting at *i* of length *l* is returned (as a value of type *substring*), if *NONE*, the suffix of *s* starting at position *i* is returned. For convenience, *substring (s,n,l)* is equivalent to *extract (s,n,SOME (l))* and *all (s)* is equivalent to *extract (s,0,NONE)*, returning the substring corresponding to the whole string. The function *base* returns basic information on the substring, that is the original string *s*, the starting position of the substring in *s* and its length. The function *string* creates a new string by copying the characters in the substring. Functions *size* and *isEmpty* respectively return the size of the substring

```
signature SUBSTRING = sig
     structure String : STRING
     type substring
     val base : substring -> (String.string * int * int)
     val string : substring -> String.string
     val extract : (String.string * int * int option) -> substring
     val substring : (String.string * int * int) -> substring
     val all : String.string -> substring
     val isEmpty : substring -> bool
     val getc : substring -> (String.Char.char * substring) option
     val first : substring -> String.Char.char option
     val triml : int -> substring -> substring
     val trimr : int -> substring -> substring
     val slice : (substring * int * int option) -> substring
     val sub : (substring * int) -> char
     val size : substring -> int
     val concat : substring list -> String.string
     val explode : substring -> String.Char.char list
     val isPrefix : String.string -> substring -> bool
     val compare : (substring * substring) -> order
     val collate : ((String.Char.char * String.Char.char) -> order) -> (substring * substring) -> orde
     val splitl : (String.Char.char -> bool) -> substring -> (substring * substring)
     val splitr : (String.Char.char -> bool) -> substring -> (substring * substring)
     val splitAt : (substring * int) -> (substring * substring)
     val dropl : (String.Char.char -> bool) -> substring -> substring
     val dropr : (String.Char.char -> bool) -> substring -> substring
     val takel : (String.Char.char -> bool) -> substring -> substring
     val taker : (String.Char.char -> bool) -> substring -> substring
     val position : String.string -> substring -> (substring * substring)
     val span : (substring * substring) -> substring
     val translate : (String.Char.char -> String.string) -> substring -> String.string
     val tokens : (String.Char.char -> bool) -> substring -> substring list
     val fields : (String.Char.char -> bool) -> substring -> substring list
     val foldl : ((String.Char.char * 'a) -> 'a) -> 'a -> substring -> 'a
     val foldr : ((String.Char.char * 'a) -> 'a) -> 'a -> substring -> 'a
     val app : (String.Char.char -> unit) -> substring -> unit
end
```

Figure 4.10: The signature *SUBSTRING*

and whether the substring is empty.

Some functions provide the same functionality as functions in the *String* structure, but at the level of substrings: *sub* returns the character at a given position of the substring (position 0 being the first character of the substring), *concat* returns a new string made up of the concatenation of a list of substrings, *explode* returns the list of characters in a substring, *isPrefix* reports whether a given string is a prefix of a substring, *compare* does substring comparison using the standard lexicographic ordering, *collate* does the same but using a lexicographic order derived from a custom ordering on characters (see Section 4.2), *tokens* and *fields* work as for strings (but return substrings), and so does *translate* (which creates a new string). Operations *foldl*,*foldr* and *app* are standard functions for aggregate types, which we'll see in more details in the next section when we talk about lists.

The rest of the functions are specific to substrings. To get substrings of substrings, one can use *triml* or *trimr*, which remove *k* characters from the given substring (from the left or the right, resepctively), returning an empty substring if not enough characters are present. More generally, *slice* will extract a substring from a substring, just as the function *extract*, but acting on substrings. The function *splitAt* applied to a substring *ss* returns the pair of substrings made up respectively of the first *i* characters of *ss* and the rest, respectively.

A different class of functions to get at substrings of substrings are the *splitl* and *splitr* functions. They take a predicate on characters, and scan the given substring from the left and the right respectively, looking for the first character that does not satisfy the predicate. They return the split of the substring into the span up to that character (but not including it) and the rest. The pair of substrings returned always is of the form *(left part,right part)*. The derived functions *takel*, *dropl*, *taker* and *dropr* return appropriate portions of the split. In fact, *takel p s* is equivalent to *#1 (splitl p s)*, *taker* is equivalent to *#2 (splitr p s)* and similarly for *dropl* and *dropr*.

The function *position* also splits a substring *ss*, but takes another string *s* as input and returns the pair *(pref,suff*) where *suff* is the longest suffix of *ss* with the string *s* as a prefix — *pref* being the first part of the substring *ss*, up to the leftmost point where the string *s* can be found in the substring.

The function *span* is more complicated. Given two substrings of the same string, it returns the substring made up of the two substrings and every character of the underlying string in between. The exception *Span* is raised if the substrings do not share the same underlying string or if the first substring is not to the left of the second substring in the underlying string.

```
structure StringCvt : sig
  datatype radix = BIN | OCT | DEC | HEX
  datatype realfmt
  = SCI of int option
  | FIX of int option
  | GEN of int option
  | EXACT
  type ('a, 'b) reader = 'b -> ('a * 'b) option
  val padLeft : char -> int -> string -> string
  val padRight : char -> int -> string -> string
  val splitl : (char -> bool) -> (char, 'a) reader ->'a -> (string * 'a)
  val takel : (char -> bool) -> (char, 'a) reader ->'a -> string
  val dropl : (char -> bool) -> (char, 'a) reader ->'a -> 'a
  val skipWS : (char, 'a) reader -> 'a -> 'a
  type cs
  val scanString : ((char, cs) reader -> ('a, cs) reader) -> string -> 'a option
end
```

Figure 4.11: The structure *StringCvt*

String conversions

The structure *StringCvt* (signature in Figure 4.11) handles the basics of string conversion, and is the focus that leads us to another convention of the Basis. Notice before anything else that *StringCvt* defines the constants for number radices and for the formatting of real numbers. Also, it provides functions *padLeft* and *padRight* that insert the supplied character respectively on the left or on the right of the given string until the target length of the string has been achieved.

As we have seen earlier, the Basis comes with a set of design conventions, ways of uniformly handling various kind of data. For example, the function *compare* is provided for types admitting comparisons of their elements, and *toString* and *fromString* are provided for most types to handle conversion to and from strings. We now introduce a new convention, to handle the reading of values of different types from generic streams of values. The key ingredient is given by the *reader* type declared in *StringCvt*:

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

A value of type *('a,'b) reader*, intuitively, is a function that takes as input a "stream" of type *'b* and attempts to read a value of type *'a* from it. It returns *SOME (v,ns)* with *v* a value of type *'a* read from the stream and *ns* the rest of the stream (after *v* has been read off). It returns *NONE* if no value of type *'a* can be read from the stream. To keep descriptions short, we often talk about an *'a reader*, to mean a value of type *('a,'b) reader* for some unspecified stream type *'b*.

An important class of readers are derived from character readers over various streams. A function that takes a character reader and returns an *T reader* for some type *T* is called a scanning function for type *T*. Intuitively, scanning functions create a *T reader* by reading off characters from the stream until an appropriate value of type *T* has been built. For a given type *T*, a scanning function has type:

```
(char,'b) reader -> (T,'b) reader.
```

Most types in the Basis provide such a scanning function, typically called *scan*. For example, *Int.scan* which has type:

```
(char,'b) reader -> (int,'b) reader.
```

(Note that this type is sometimes reported by expanding the abbreviation of the second reader: *(char,'b) reader* $\rightarrow$*'b* $\rightarrow$*(int,'b) option*). The main reasons for deriving general readers from character readers is that this operation is a generalization of converting a value from a string (indeed, as we shall see, *fromString* functions are typically derived from *scan* functions), and because character readers are the most common kind of readers. Character readers can be derived easily from any character source, which include strings, substrings, and most importantly input streams (we will return to input stream in Section 4.5).

Time to look at some examples. Although strings are a natural starting point, they make for a slightly messy example. So let's start with substrings. The *Substring* structure contains a function *getc* of type *substring* $\rightarrow$*(char* $\times$*substring) option*, which one recognizes as the type *(char,substring) reader*, a character reader. Indeed, if we apply the reader to a substring, we should get a character along with the remainder of the substring. This is easily verified:

```
Substring.getc (Substring.all "hello");
val it = SOME (#"h",-) : (char * substring) option
- Substring.string (#2 (valOf (it)));
val it = "ello" : string
```

Although strings don't come equipped with a character reader, we can fashion one from the above:

```
fun stringGetc (s) = let
  val ss = Substring.all (s)
in
  case Substring.getc (ss)
    of NONE => NONE
     | SOME (c,ss') => SOME (c,Substring.string (ss'))
end
```

and indeed,

```
- stringGetc ("hello");
val it = SOME (#"h","ello") : (char * string) option
```

Scanning functions create new readers from existing readers. For instance, the function *Int.scan* takes a character reader into an integer reader. Consider the function

```
val stringGetInt = Int.scan StringCvt.DEC stringGetc
```

This should give us an integer reader over strings, that is a function to read off decimal integers from strings. This is again easily verified:

```
- stringGetInt ("1020hello");
val it = SOME (1020,"hello") : (int * string) option
- stringGetInt ("  1020hello");
val it = SOME (1020,"hello") : (int * string) option
- stringGetInt "foo";
val it = NONE : (int * string) option
```

As the last line shows, if the reader fails to read for whatever reason, the value *NONE* is returned. Notice that the readers created by *Int.scan* skip over any leading whitespace, so you don't have to worry about it. What if you wanted to explicitely skip whitespaces? (For example, if you were yourself writing a scanning function?). Time to make a diversion back to some of the functions provided by *StringCvt*. The function *skipWS* take a *(char,'b) reader* and a stream of type *'b* and returns a new stream of type *'b*; the idea being that the stream produce has all of its leading whitespaces removed. The reader provided is used to read off the characters of the stream to remove those whitespaces (defined by the *Char.isSpace* predicate).

More generally, the functions *splitl*, *takel* and *dropl* in *StringCvt* handle character streams by taking a predicate on characters, a character reader for the stream, and a stream, and returning respectively a pair consisting of the string made up of all the consecutive characters from the stream that matched the predicate and the rest of the stream, or just the matching string or just the remainder of the stream. In fact, *skipWS* is equivalent to *dropl Char.isSpace*.

So let's recap: a reader takes a stream and returns an element read from the stream and the remainder of the stream. A scanning function for a given type takes a character reader and converts it to a reader for that type. To use a reader, one simply applies it to an appropriate stream. We have seen readers from substrings, and we have built readers from strings. In the following sections, we will see readers from lists and vectors. Reading a value from a string via a scanning function is so common that a function is available in *StringCvt* to greatly automate the process. The function *scanString* takes a scanning function for a type *T* and a string, and scans the string, trying to read a value of type *T* using the supplied scanning function. Note that we do not need to supply a character reader function (to feed to the scanning function). One is implicitely created to read characters from the string. Here is a suitable implementation of *scanString*:

```
fun scanString f s = let
  val ss = Substring.all
  val scan = f (Substring.getc)
in
  case (scan ss)
    of NONE => NONE
     | SOME (v,_) => SOME (v)
end
```

Notice the result returned by *scanString*: in the case of a successful scan, it does not return the remainder of the string after the match. The reason for this it to get the right result for the various *fromString* functions, of which *scanString* is a generalization. Indeed, *fromString* can easily be obtained by a simple composition. For instance, *Int.fromString* is equivalent to *StringCvt.scanString (Int.scan StringCvt.DEC)*, *Bool.fromString* is equivalent to *StringCvt.scanString Bool.scan*, and so on. If one is interested in the rest of the string following the scan, one can use a translation to substrings directly, or pass the function *stringGetc* defined earlier to the appropriate scanning function to get a reader over strings.

Reades are general and provide a very convenient mechanism for reading values from streams. In Chapter **??**, we will introduce a new type of character source for which character readers can be defined, namely input streams. On the other hand, in Chapter **??**, we will see how to define scanning functions for general regular expressions, taking character readers into string readers.

4.4 Aggregate types

In Chapter 2, we have seen how lists are an important data type, allowing you to put together an arbitrary number of values of the same type. In this chapter, we explore the support of the Basis with respect to a wider class of such aggregate types, including lists, but also vectors and arrays.

The structure *List* (whose signature is given in Figure 4.12) provides many useful functions to handle lists, as well as defining the *list* type itself. The exception *Empty* is raised by functions that expect a non-empty list and are given an empty list as an argument (such as *hd* or *tl*). The functions *null*, *hd*, *tl*, *lengh* and @ (for concatenation) have already been presented in Chapter 2 — they are exported to top-level from this structure. As usual, @ is infix at toplevel, but must be used in its prefix form if qualified, e.g. *List.@ (l1,l2)*. Some functions are rather clear from their name, so that *last* returns the last element of a list, *nth* the n^{th} element of the list (counting from 0 as the first element), *take* and *drop* respectively return the first *n* elements of the list as a list and the rest of the list after the first *n* elements, *rev* reverses a list, while *concat* takes a list of lists and concatenates them all in their given order (so that *List.@ (l1,l2)* is equivalent to *List.concat [l1,l2]*. The

```
structure List : sig
  datatype list = datatype list
  exception Empty
  val null : 'a list -> bool
  val length : 'a list -> int
  val @ : ('a list * 'a list) -> 'a list
  val hd : 'a list -> 'a
  val tl : 'a list -> 'a list
  val last : 'a list -> 'a
  val getItem : 'a list -> ('a * 'a list) option
  val nth : ('a list * int) -> 'a
  val take : ('a list * int) -> 'a list
  val drop : ('a list * int) -> 'a list
  val rev : 'a list -> 'a list
  val concat : 'a list list -> 'a list
  val revAppend : ('a list * 'a list) -> 'a list
  val app : ('a -> unit) -> 'a list -> unit
  val map : ('a -> 'b) -> 'a list -> 'b list
  val mapPartial : ('a -> 'b option) -> 'a list -> 'b list
  val find : ('a -> bool) -> 'a list -> 'a option
  val filter : ('a -> bool) -> 'a list -> 'a list
  val partition : ('a -> bool) -> 'a list -> ('a list * 'a list)
  val foldl : (('a * 'b) -> 'b) -> 'b -> 'a list -> 'b
  val foldr : (('a * 'b) -> 'b) -> 'b -> 'a list -> 'b
  val exists : ('a -> bool) -> 'a list -> bool
  val all : ('a -> bool) -> 'a list -> bool
  val tabulate : (int * (int -> 'a)) -> 'a list
end
```

Figure 4.12: The structure *List*

function *revApend (l1,l2)* is equivalent to *(rev l1)@l2)*, useful for some patterns of recursive functions.

The remaining functions perform operations while walking down a list. We have already seen *map* and *app*. The function *mapPartial* is similar to *map*, but takes in a function returning an option value, and accumulates only the non-*NONE* results into a list. In our previous terminology, it takes a partial function and returns the accumulated results for the defined elements only. The function *find* takes a predicate *p* and searches a list for the first value *v* for which *p (v)* evaluates to *true*, and returns *SOME (v)*; it returns *NONE* if such a value cannot be found. The function *filter* also takes a predicate *p* and returns the list of all elements *v* for which *p (v)* evaluates to *true*. The function *partition* is even more general, returning a pair of lists *(pos,neg)*, where *pos* is the list of elements *v* for which *p (v)* is *true*, and *neg* is the list of elements *v* for which *p (v)* is *false*,. The relative order of elements in *pos* and *neg* is that of the original list.

Functions *exists* and *all* (both taking a predicate *p*) check if at least one element *v* is such that *p (v)* is *true* and all the elements *v* are such that *p (v)* is *true*. The operations are short-circuiting; as soon as an element evaluating to *true* is encountered, *exists* returns *true*, while as soon as an element evaluating to *false* is encountered, *all* returns *false*. The function *exists* is a generalization of *orelse* and *all* is a generalization of *andalso*. The function *tabulate* is used to construct a list from a function: *tabulate (n,f)* returns the list *[f(0),...,f(n-1)]*.

The functions *foldl* and *foldr* are the left and right folding operations, returning

$$f(x_n, f(x_{n-1}, ..., f(x_1, b)))$$

and

$$f(x_1, f(x_2, ..., f(x_n, b)))$$

respectively when applied to a value *b* and list *[x1,...,xn]*. Clearly, the result is the same if *f* is commutative. These functions are deceptively simple, and are cornerstones of functional programming using lists (and, it turns out, using other data types). For instance, *map* is equivalent to *foldr (op ::) []*, *filter* is equivalent to *foldr (fn (x,r) => if p (x) then x::r else r) []*. More complex still, the function *partition* is equivalent to:

```
foldr (fn (x,(pos,neg)) => if p (x) then (x::pos,neg) else (pos,x::neg)) ([],[]).
```

The key point is that since many operations can be defined as a *foldl* or a *foldr*, any data type for which folding operations can be meaningfully defined can use such derived operations. For example, vectors and arrays, which we'll see next, define folding operations, as do hash tables in the SML/NJ Library (see Chapter 7).

```
structure ListPair : sig
  val zip : ('a list * 'b list) -> ('a * 'b) list
  val unzip : ('a * 'b) list -> ('a list * 'b list)
  val map : ('a * 'b -> 'c) -> ('a list * 'b list) -> 'c list
  val app : ('a * 'b -> unit) -> ('a list * 'b list) -> unit
  val foldl : (('a * 'b * 'c) -> 'c) -> 'c -> ('a list * 'b list) -> 'c
  val foldr : (('a * 'b * 'c) -> 'c) -> 'c -> ('a list * 'b list) -> 'c
  val all : ('a * 'b -> bool) -> ('a list * 'b list) -> bool
  val exists : ('a * 'b -> bool) -> ('a list * 'b list) -> bool
end
```

Figure 4.13: The structure *ListPair*

The last function in *List* is *getItem*, which views a list as a stream, in the sense of Section 4.3. Once one sees the type of *getItem*, one recognizes it as a *('a,'a list) reader*, so that given an *'a list*, *getItem* is an *'a* value reader. This provides the first "natural" reader for streams which is not character-based. Of course, if one is given a list of characters, one can use standard character reader functions and scanning functions. Although it is not the most efficient approach, we could implement *stringGetc* as:

```
fun stringGetc' (s) = let
  val l = String.explode (s)
in
  case List.getItem (l)
    of NONE => NONE
     | SOME (c,l') => SOME (c,String.implode (l'))
end
```

Operations involving pairs of lists are common enough to warrant a structure *ListPair* in the Basis, with a signature given in Figure 4.13. The fundamental operation on pairs of lists is *zip*, which takes two lists *[x1,x2,...]* and *[y1,y2,...]*, and produces a new list with the corresponding elements paired, *[(x1,y1),(x2,y2),...]*. The length of the resuling list is the length of the shortest one, the extra elements of the longer list being ignored. The function is itself easy to implement:

```
fun zip ([],_) = []
  | zip (_,[]) = []
  | zip (x::xs,y::ys) = (x,y)::zip(xs,ys)
```

The function *unzip* performs the inverse operation, taking a list of pairs and splitting each pair apart. It is also easy to define:

```
fun unzip [] = ([],[])
  | unzip ((x,y)::r) = let
      val (xl,yl) = unzip (r)
    in
      (x::xl,y::yl)
    end
```

```
structure Vector : sig
  eqtype 'a vector
  val maxLen : int
  val fromList : 'a list -> 'a vector
  val tabulate : (int * (int -> 'a)) -> 'a vector
  val length : 'a vector -> int
  val sub : ('a vector * int) -> 'a
  val extract : ('a vector * int * int option) -> 'a vector
  val concat : 'a vector list -> 'a vector
  val mapi : ((int * 'a) -> 'b) -> ('a vector * int * int option) -> 'b vector
  val map : ('a -> 'b) -> 'a vector -> 'b vector
  val appi : ((int * 'a) -> unit) -> ('a vector * int * int option) -> unit
  val app : ('a -> unit) -> 'a vector -> unit
  val foldli : ((int * 'a * 'b) -> 'b) -> 'b -> ('a vector * int * int option) ->
    'b
  val foldri : ((int * 'a * 'b) -> 'b) -> 'b -> ('a vector * int * int option) ->
    'b
  val foldl : (('a * 'b) -> 'b) -> 'b -> 'a vector -> 'b
  val foldr : (('a * 'b) -> 'b) -> 'b -> 'a vector -> 'b
end
```

Figure 4.14: The structure *Vector*

Most other functions in the *ListPair* structure can be derived from *zip* (although they may be more efficiently implemented directly). The functions *map* and *app* do as expected, but these operations expect a function taking two elements, one from each list. In fact, *ListPair.map f (l1,l2)* is equivalent to *List.map f (zip (l1,l2))*, and similarly for *app*. Moreover, the function *foldl f b (l1,l2)* is equivalent to *List.foldl (fn ((a,b),c) => f (a,b,c)) b (zip (l1,l2))* and similarly for *foldr*. The functions *all p (l1,l2)* and *exists p (l1,l2)* are equivalent to *List.all p (zip (l1,l2))* and *List.exists p (zip (l1,l2))* respectively.

Vectors, just as lists, are used to implement sequences of elements of the same type. The difference is that vectors provide more efficient access to elements in the middle of the sequence, at the cost of a loss of flexibility while constructing vectors. Just as with lists, vector elements are immutable (unless one constructs a vector of reference cells explicitely, of course). Arrays, which will be discussed next, allow one to modify elements in place. Many vector operations are similar to operations on lists and on strings. Vectors come in fundamentally two flavors, polymorphic vectors and monomorphic vectors. Polymorphic vectors, defined in a structure *Vector* whose signature is given in Figure 4.14, define a type *'a vector*, a vector of elements of type *'a*, whereas monomorphic vectors are defined in specific structures that specify what the type of elements of the vectors is, for example structure *IntVector* defining a type *IntVector.vector* of integer vectors. The main reason for the distinction is one of efficiency: the compiler can use a much

Figure 4.15: The structure *Subvector*

more efficient layout for, say, *ByteVector.vector* than the general *'a vector*. But the underlying operations are the same. We discuss here polymorphic vectors. Monomorphic vectors in SML/NJ include *CharVector*, *Word8Vector*, *RealVector*.

The value *maxLen* returns the maximum length of any vector supported by the implementation. The exception *Size* is raised if this limit is ever reached (say if one attempts to create a vector of length *maxLen+1*). The function *length* returns the length of a vector, while *fromList* and *tabulate* create new vectors, respectively from a list of elements and from a function (as in the case of *List.tabulate*). The function *concat* concatenates a list of vectors (in order) into a single vector. The function *sub* returns the element at the given index in the vector, the index of the first element being 0. Standard operations *foldl*, *foldr*, *map* and *app* are provided, with the expected semantics. As we mentioned earlier, the fact that one can fold over the elements of a vector means that we can implement a whole panoply of operations, such as filtering and partitioning operations.

An interesting aspect of vectors is reminiscent of strings, and it is the notion of a vector slice, akin to a substring. A vector slice is described by a tuple *(v,i,opt)* where *v* is a vector, *i* is an index (the starting point of the slice) and *opt* is an option value giving the length of the slice as *SOME (n)* or *NONE* for the end of the vector. A slice can be extracted into its own vector by the function *extract*. More interestingly, functions *foldli*, *foldri*, *mapi* and *appi* are provided to apply *foldl*, *foldr*, *map* and *app* operations only to the elements in the given slice. As an extra generality, the functions passed as an argument to these operations are passed as a first argument the index of the element. The less general *foldl*, *foldr*,*map* and *app* operations can easily be derived from the general ones. For example, *foldl f init v* is equivalent to *foldli (fn (_,a,x) => f (a,x)) init (v,0,NONE)*. Finally, although no function *toList* is provided, it can be easily synthesized as:

```
fun toList (v) = foldlr (op ::) [] vec
```

After our discussion of readers and streams in Section 4.3, and our remark about lists being treated as streams through the function *List.getItem*, one notices that no such support exists in the Basis for vectors. Similarly, no such support existed for strings either, and for the same reason: efficiency. For strings, one could define readers by going through substrings. Although no structure corresponding to subvectors exists, they can be easily defined, and provide an interesting exercise.

Let us provide the basic functionality (and give us some insight into the implementation of the *Substring* structure). We define a structure *Subvector* with a signature given in Figure 4.15 (note that it could easily be extended to all that *Substring* defines). The type *subvector* is easily defined as a vector slice:

```
type 'a subvector = ('a vector * int * int option)
```

The functions *all* and *vector* are also clear:

```
fun all (v) = (v,0,NONE)
fun base (sv) = sv
fun vector (sv) = Vector.extract (sv)
```

We could push in *map*, *app* and folding functions easily, from the *mapi*, *appi* and folding functions on vector slices in *Vector*. We concentrate on *getElem*, the reader for subvectors:

```
fun getElem (v,i,NONE) = if (i<length (v))
                           then SOME (sub (v,i),(v,i+1,NONE))
                         else NONE
  | getElem (v,i,SOME (0)) = NONE
  | getElem (v,i,SOME (n)) = if (i<length (v))
                               then SOME (sub (v,i),i+1,SOME (n-1))
                             else NONE
```

Using these functions, we can rather easily write the (inefficient) *vectorGetElem* function, mimicking the *stringGetc* function:

```
fun vectorGetElem (v) = let
  val sv = Subvector.all (v)
in
  case (Subvector.getElem (sv))
    of NONE => NONE
     | SOME (e,sv') => SOME (e,Subvector.vector (sv'))
end
```

Even though this function is of dubious use, it is instructive to show we can build the infrastructure for streams and readers even for types for which it is not provided.

Finally, we come to arrays. Arrays are, like vectors, sequences of elements that allow for efficient access to elements "in the middle", at the same cost in flexibility in constructing arrays. Arrays moreover admit operations to modify array elements in place, without having to reconstruct a new array. Arrays also come in both polymorphic and monomorphic forms, and again here we describe polymorphic arrays. Monomorphic arrays provided by SML/NJ include *CharArray*, *Word8Array*, *RealArray*.

Polymorphic arrays are implemented by a structure *Array* whose signature is given in Figure 4.16. The signature is similar to that of polymorphic vectors, including most of the same operations, down to the notion of an array slice, the array equivalent of a vector slice. We discuss here the differences.

```
structure Array : sig
  eqtype 'a array
  type 'a vector
  val maxLen : int
  val array : (int * 'a) -> 'a array
  val fromList : 'a list -> 'a array
  val tabulate : (int * (int -> 'a)) -> 'a array
  val length : 'a array -> int
  val sub : ('a array * int) -> 'a
  val update : ('a array * int * 'a) -> unit
  val extract : ('a array * int * int option) -> 'a vector
  val copy : {src : 'a array, si : int, len : int option, dst : 'a array, di :
    int} -> unit
  val copyVec : {src : 'a vector, si : int, len : int option, dst : 'a array, di
    : int} -> unit
  val appi : ((int * 'a) -> unit) -> ('a array * int * int option) -> unit
  val app : ('a -> unit) -> 'a array -> unit
  val foldli : ((int * 'a * 'b) -> 'b) -> 'b -> ('a array * int * int option) ->
    'b
  val foldri : ((int * 'a * 'b) -> 'b) -> 'b -> ('a array * int * int option) ->
    'b
  val foldl : (('a * 'b) -> 'b) -> 'b -> 'a array -> 'b
  val foldr : (('a * 'b) -> 'b) -> 'b -> 'a array -> 'b
  val modifyi : ((int * 'a) -> 'a) -> ('a array * int * int option) -> unit
  val modify : ('a -> 'a) -> 'a array -> unit
end
```

Figure 4.16: The structure *Array*

The main differences take advantage of updatability. The function *array (n,a)* creates a new array of length *n* with all positions initialized to the value *a*. The function *update* updates the value at index *i* of the array to the specified value. Also, note that *extract* returns a vector of the elements in the array slice, not an array.

The functions *copy* and *copyVec* copy the elements of the array slice (or vector slice respectively) specified by *(src,si,len)* into array *dst*, starting at position *di* (that is, element at position *si* in *src* is put at position *di* in *dst*, and so on). An exception *Subscript* is raised if the source slice is not valid, or if it does not fit at the given position in the destination array.

The functions *modify* and *modifyi* apply a transformation function to the elements of the array, replacing them in place. As in the *map*/*mapi* or *app*/*appi* variation, *modify* takes a standard transformation function and applies it to the whole array, while *modifyi* applies the transformation function to an array slice, passing in the index along with the element to be transformed.

As with vectors, converting an array to a list is achieved with the expression *foldr (op ::) []*. We can also derive readers from arrays as we did for vectors, although they are much less useful. Defining a reader on a stream for which arbitrary elements can be changed leads to unpleasant semantics for functions using the reader. For instance, one cannot look ahead in a "safe" way.

A variation on arrays, two-dimensional arrays, are also provided in the Basis. Again, they exist in polymorphic and monomorphic forms. We describe polymorphic two-dimensional arrays as provided by the structure *Array2*, whose signature is given in Figure 4.17. The functions are similar enough to those in *Array* that they should not require too much explanation.

The functions *array*, *fromList*, *sub*, *update* perform the operations equivalent to their *Array* counterparts. New functions *dimensions*, *nCols* and *nRows* return the dimensions, number of columns and number of rows of an array, respectively. The functions *row* and *column* extract a given row or column into a vector of elements.

Instead of defining a slice, two-dimensional arrays define a region, given by a record {*base,row,col,nrows,ncols*}, where *base* is the underlying two-dimensional array, *(row,col)* the point where the region starts, and *nrows* and *ncols* the number of rows and columns starting from the point making up the region (as in the case of slices, a value of *NONE* for *nrows* or *ncols* signifies until the last row or column). The function *copy*, in analogy with its *Array* counterpart, copies a region of an array into another array, at the specified destination point.

For functions requiring the traversal of a two-dimensional array, such as *tabulate*, *app*, *appi*, *modify*, *modifyi* and the folding operations, an indication is expected for the order in which the array is to be traversed. The type *traversal* defines the possible orders: *RowMajor* indicates that the array should be traversed row-by-

```
structure Array2 : sig
  eqtype 'a array
  type 'a region = {base : 'a array, row : int, col : int, nrows : int option,
    ncols : int option}
  datatype traversal
    = RowMajor
    | ColMajor
  val array : (int * int * 'a) -> 'a array
  val fromList : 'a list list -> 'a array
  val tabulate : traversal -> (int * int * ((int * int) -> 'a)) -> 'a array
  val sub : ('a array * int * int) -> 'a
  val update : ('a array * int * int * 'a) -> unit
  val dimensions : 'a array -> (int * int)
  val nCols : 'a array -> int
  val nRows : 'a array -> int
  val row : ('a array * int) -> 'a Vector.vector
  val column : ('a array * int) -> 'a Vector.vector
  val copy : {src : 'a region, dst : 'a array, dst_row : int, dst_col : int} ->
    unit
  val appi : traversal -> ((int * int * 'a) -> unit) -> 'a region -> unit
  val app : traversal -> ('a -> unit) -> 'a array -> unit
  val modifyi : traversal -> ((int * int * 'a) -> 'a) -> 'a region -> unit
  val modify : traversal -> ('a -> 'a) -> 'a array -> unit
  val foldi : traversal -> ((int * int * 'a * 'b) -> 'b) -> 'b -> 'a region -> 'b
  val fold : traversal -> (('a * 'b) -> 'b) -> 'b -> 'a array -> 'b
end
```

Figure 4.17: The structure *Array2*

row, and *ColMajor* indicates that the array should be traversed column-by-column. The semantics of the traversal functions are as their *Array* counterparts, aside from the issue of the order of traversal.

4.5 Input and output

Having described the types support in the Basis, we can now turn to the second major role of the Basis, the interoperation with the underlying operating system. In this section, we focus on the support for textual input and output, although we will only skim the subject. A much more in-depth description of the input and output support will be given in Chapter **??**, where we showcase the flexibility of the infrastructure.

The structure *TextIO* provides support for imperative text-based input and output. It is imperative, because reading an input stream modifies the stream, and it is text-based in that reading and writing is based on characters and strings. In Chapter **??**, we shall see input and output functions based on a functional view of streams, and input and output functions at the conceptual level of system calls to the operating system. The signature for *TextIO* is given in Figure 4.18.

The key types of the structure are *instream* and *outstream*, the types of imperative streams for input and output, respectively. We discuss input and output operations separately. We note that streams are buffered, meaning that input and output from the actual device is performed in blocks of an unspecified size. It also means that even if one calls output functinos multiple times, the output may not be performed on the device right away, only when the internal buffer fills up. Forcing the buffer content to the device before the buffer is full is commonly called flushing. Input streams have a special state called end-of-stream (or end-of-file, EOF). EOF is reached when the stream cannot supply any more characters. This may happen because the end of a file has been reached, for example. Note that EOF state need to be permanent. If a file is updated by another process while the program is reading from it and is at EOF, a subsequent input may well succeed and deliver more characters.

Obtaining an input stream for input can be done in various ways: the value *stdIn* holds the current standard input stream, by default hooked to the standard input of the terminal or window where the SML interactive loop is running. The functions *openIn* and *openString* create an input stream from a file and from a string, respectively. An input stream opened from a string will serve the characters from the string in left to right order. Once an input stream is done with, it should be closed by a call to *closeIn*. Various functions for reading characters and strings are provided. The basic input function is *input*, which attempts to read a string of

```
structure TextIO : sig
  structure StreamIO : STREAM_IO
  type vector = StreamIO.vector
  type elem = StreamIO.elem
  type instream
  type outstream
  val input : instream -> vector
  val input1 : instream -> elem option
  val inputN : (instream * int) -> vector
  val inputAll : instream -> vector
  val canInput : (instream * int) -> int option
  val lookahead : instream -> elem option
  val closeIn : instream -> unit
  val endOfStream : instream -> bool
  val output : (outstream * vector) -> unit
  val output1 : (outstream * elem) -> unit
  val flushOut : outstream -> unit
  val closeOut : outstream -> unit
  val getPosIn : instream -> StreamIO.in_pos
  val setPosIn : (instream * StreamIO.in_pos) -> unit
  val mkInstream : StreamIO.instream -> instream
  val getInstream : instream -> StreamIO.instream
  val setInstream : (instream * StreamIO.instream) -> unit
  val getPosOut : outstream -> StreamIO.out_pos
  val setPosOut : (outstream * StreamIO.out_pos) -> unit
  val mkOutstream : StreamIO.outstream -> outstream
  val getOutstream : outstream -> StreamIO.outstream
  val setOutstream : (outstream * StreamIO.outstream) -> unit
  structure StreamIO : TEXT_STREAM_IO
  val inputLine : instream -> string
  val outputSubstr : (outstream * substring) -> unit
  val openIn : string -> instream
  val openOut : string -> outstream
  val openAppend : string -> outstream
  val openString : string -> instream
  val stdIn : instream
  val stdOut : outstream
  val stdErr : outstream
  val print : string -> unit
  val scanStream : ((Char.char,StreamIO.instream) StringCvt.reader ->
    ('a,StreamIO.instream) StringCvt.reader) -> instream -> 'a option
end
```

Figure 4.18: The structure *TextIO*

characters from the input stream. It is unspecified how much of the stream is read. Alternatively, *inputN* attempts to read at most *N* chars from the stream, maybe less if EOF is reached. The function *inputAll* will attempt to read the whole stream, until EOF is reached. Finally, *input1* attempts to read a single character, returning *SOME (c)* if one is found, *NONE* if EOF is reached.

In all of the above cases, the call may block (or an *IO.Io* exception raised) if no data is available from the input stream — that is, EOF has not been reached, but no data is available, because say another process is slowly writing to the stream. For some types of streams, it may be possible to check if the stream can provide a certain number of characters without blocking, and that's what *canInput* tries to report. It reports how many characters can be read immediately without blocking, or *NONE* if blocking occurs immediately (*SOME (0)* indicates EOF). On a related note, *lookahead* returns an element from the stream if one is available (or *NONE* if at EOF), just like *input1*, but it does not remove the character from the stream, so that other input operations can retrieve the character. Implementing a lookahead of more than one character can be done with the underlying functional stream subsystem. This will be covered in Chapter **??**. The function *endOfStream* returns *true* if and only if the stream is at EOF. Finally, *inputLine* returns a line from the stream, namely the string of characters up to and including the first newline character. If EOF is reached before a newline, a newline character is appended at the end of the string anyways. If at EOF, *inputLine* returns the empty string.

Output works in a similar way. The values *stdOut* and *stdErr* hold the current output and error streams respectively, typically hooked to the terminal or window holding the SML interactive loop. The functions *openOut* and *openAppend* open a file for output, respectively overwriting it or appending to it.

The actual output is performed by the functions *output* and *output1*, which send a string or a single character to the specified output stream. The function *outputSubstring* can be used to send a substring of a string, and should be more efficient than the equivalent *output o Substring.string*.

Because output is buffered, it may be necessary to use *flushOut* to flush the stream. The function *print* sends its argument to *stdOut* and flushes the stream automatically. It is also exported to toplevel for convenience. Finally, as in the case of input, output streams should be closed through *closeOut* when done with.

It is straightforward to use the functions we have seen. Consider the following example, implementing a simple query-reply loop: an input string is queried from the user, it is processed, a reply is output. Since we don't want to pin down where the input is taken from or where the output is sent, or even what processing is performed, we parametrize the routine by all of those bits of information:

```
fun queryReply (ins,outs,process,prompt) = let
  fun loop () = (case TextIO.inputLine (ins)
                  of "" => ()
                   | s => let
                            val result = process (s)
                         in
                            TextIO.output (outs,s);
                            TextIO.output1 (outs,#"\n");
                            TextIO.flushOut (outs);
                            loop ()
                         end)
in
  loop ()
end
```

Finally, it seems logical that input streams should provide the perfect backdrop for the readers and steams approach we have seen in Section 4.3. Because of various trickiness dues to the nature of imperative I/O, to which we will return in Chapter **??**, using an imperative stream as a reader stream is done via a function *scanStream* similar in spirit to *scanString*. The function *scanStream* takes a scanning function (which you may recall converts a character reader into a value reader for some type) and an input stream, and attempts to read a value from the stream using the reader. If successful, the value read is returned and the stream is in a position after the value read. If no value can be successfully read, *NONE* is returned and the stream is left untouched.

As a technicality, notice that the scanning function must work on *StreamIO.instream* types, which we discuss in Chapter **??**. Since all the scanning functions we have seen until now are polymorphic over the character source, we are unaffected by this restriction for the time being.

To illustrate this, let us generalize the previous query-reply function to take a scanning function as an argument, and processing the provided value instead of just a string:

```
fun queryReply' (ins,outs,process,scan) = let
  fun loop () = (case TextIO.scanStream scan ins
                  of NONE => ()
                   | SOME (v) => let
                                   val result = process (v)
                                 in
                                   TextIO.output (outs,result);
                                   TextIO.output1 (outs,#"\n");
                                   TextIO.flushOut (outs);
                                   loop ()
                                 end)
in
  loop ()
end
```

with type:

```
val queryReply' : TextIO.instream * TextIO.outstream * ('a -> string)
                    * ((char,'b) StringCvt.reader ->
                              ('a,'b) StringCvt.reader) -> unit
```

```
structure OS : sig
  eqtype syserror
  exception SysErr of (string * syserror option)
  val errorMsg : syserror -> string
  val errorName : syserror -> string
  val syserror : string -> syserror option
  structure FileSys : OS_FILE_SYS
  structure Path : OS_PATH
  structure Process : OS_PROCESS
  structure IO : OS_IO
end
```

Figure 4.19: The structure *OS*

An interesting question: can we recuperate the original *queryReply* from the above? We need to pass in a scan function for strings! One possibility is to provide a scanner until a newline, to mimic what *inputLine* is doing:

```
fun scanTilNL cr s =
```

4.6 System functions

The structure *OS* (signature in Figure 4.19) is the starting point for a largely operating system independent model for handling resources from the underlying operating system, such as the file system, directory paths, and processes. The *OS* structure defines the exception *SysErr* that is used to report operating system error conditions. An exception raised of the form *SysErr (msg,opt)* contains a system-dependent string describing the error, as well as an optional *syserror* value, an abstract type representing a symbolic "name" for the error. Functions *errorName* and *syserror* convert between symbolic names and system-dependent names.

Four substructures hang off the *OS* structure: *FileSys*, *Path*, *Process* and *IO*. We beg off in these notes describing *OS.IO*, which provides support for polling I/O devices through explicit I/O descriptors.

The *OS.FileSys* substructure, whose signature is given in Figure 4.20, handles interactions with the file system. They raise the exception *OS.FileSys* in case of error. In general, functions which end up reading or writing files require that the appropriate permissions be set for those files and the directory that contains them.

Directory handling accounts for a a large part of the structure. The function *chDir* changes the current working directory to the specified directory (support for writing paths will be describe next, when we look at the *OS.Path* substructure), while *getDir* returns the current working directory. The initial working directory

```
structure FileSys : sig
  type dirstream
  val openDir : string -> dirstream
  val readDir : dirstream -> string
  val rewindDir : dirstream -> unit
  val closeDir : dirstream -> unit
  val chDir : string -> unit
  val getDir : unit -> string
  val mkDir : string -> unit
  val rmDir : string -> unit
  val isDir : string -> bool
  val isLink : string -> bool
  val readLink : string -> string
  val fullPath : string -> string
  val realPath : string -> string
  val modTime : string -> Time.time
  val fileSize : string -> Position.int
  val setTime : (string * Time.time option) -> unit
  val remove : string -> unit
  val rename : old : string, new : string -> unit
  datatype access_mode
    = A_READ
    | A_WRITE
    | A_EXEC
  val access : (string * access_mode list) -> bool
  val tmpName : unit -> string
  eqtype file_id
  val fileId : string -> file_id
  val hash : file_id -> word
  val compare : (file_id * file_id) -> order
end
```

Figure 4.20: The structure *OS.FileSys*

is very much system dependent. A new directory can be created by *mkDir*, and conversely can be removed by *rmDir*.

Getting a list of the files in a directory can be done via the directory stream functions. A directory stream is an abstract value which yields the names of the files in the directory one at a time. A directory stream on some directory is created by *openDir*, and should be closed by *closeDir*. The next filename in the stream is returned by *readDir*. Note that the pseudo-directory "." and ".." are filtered out, and that *readDir* returns the empty string "" after the last file name has been returned. The function *rewindDir* resets the directory stream so that the next filename *readDir* returns is the first one. One can use these functions to write a function which reads a whole directory at once, returning a list of filenames:

```
fun listDir (s) = let
  fun loop (ds) = (case OS.FileSys.readDir (ds)
                     of "" => [] before OS.FileSys.closeDir (ds)
                      | file => file::loop (ds))
  val ds = OS.FileSys.openDir (s)
in
   loop (ds) handle e => (OS.FileSys.closeDir (ds); raise (e))
end
```

Notice how we have mostly protected the above code from exceptions raised inside its body by intercepting the exceptions, closing the directory stream, and re-raising the intercepted exception. This is a general pattern that arises when we use so-called modal functions, that is functions which switch in and out of a mode (for instance, *openIn* and *closeIn*, or *openOut* and *closeOut*).

The remaining functions deal with just those filenames. Predicates *isDir* and *isLink* check if the filename given represents a directory or a symbolic link (for systems that support them). In the case of a symbolic link, *readLink* reads off the linked-to filename. Functions *fullPath* and *realPath* extract the underlying path to a filename, where *fullPath* returns the absolute path, and *realPath* acts as *fullPath* if it is given an absolute path, and otherwise returns a path relative to the current working directory. The key point behind *realPath* is that the resulting path is canonical (see later in this section, when we describe substructure *OS.Path*).

Time functions *modTime* and *setTime* respectively access and change the modification time of the given file. For *setTime*, if a time is specified, it is used as the new modification time, otherwise the current time is set in. For information on time support, see Section 4.7. The function *fileSize* returns the size (in bytes) of the given file.

A file (not a directory) can be deleted using *remove*. Attempting to delete a file on which streams are currently open lead to unspecified behavior. A filename can be changed by *rename*, taking in the *old* name and the *new* name.

As the above functions raise exceptions if the permissions on the accessed files are not set correctly, it is useful to have a way to check such permissions. The pred-

```
structure Path : sig
  exception Path
  exception InvalidArc
  val parentArc : string
  val currentArc : string
  val validVolume : isAbs : bool, vol : string -> bool
  val fromString : string -> isAbs : bool, vol : string, arcs : string list
  val toString : isAbs : bool, vol : string, arcs : string list -> string
  val getVolume : string -> string
  val getParent : string -> string
  val splitDirFile : string -> dir : string, file : string
  val joinDirFile : dir : string, file : string -> string
  val dir : string -> string
  val file : string -> string
  val splitBaseExt : string -> base : string, ext : string option
  val joinBaseExt : base : string, ext : string option -> string
  val base : string -> string
  val ext : string -> string option
  val mkCanonical : string -> string
  val isCanonical : string -> bool
  val mkAbsolute : (string * string) -> string
  val mkRelative : (string * string) -> string
  val isAbsolute : string -> bool
  val isRelative : string -> bool
  val isRoot : string -> bool
  val concat : (string * string) -> string
  val toUnixPath : string -> string
  val fromUnixPath : string -> string
end
```

Figure 4.21: The structure *OS.Path*

icate *access* checks if the given file has the given permissions, where a permission is a value in a datatype *access_mode* and is either *A_READ*, *A_WRITE* or *A_EXEC* (for read, write and execute permission, respectively).

The function *tmpName* generates a temporary filename which is not the name of a currently existing file.

For uniquely identifying files, the Basis provides the notion of a file identifier: given a path to a file system object, *fileId* returns a *file_id* for that object. The guarantee is that different paths yielding the same file_id mean that the path point to the same underlying objcet. Given a file_id, *hash* returns a corresponding hash value in the form of a word. Finally, *compare* is the ordering function on file_id's, in some arbitrary underlying linear ordering of the file_id's. Note that file_id's are not guaranteed persistent over underlying file system operations, such as mounting and unmounting file system devices.

As the above functions (and some in *TextIO*) need to work with paths to files, and as different operating systems have different conventions for dealing with paths, we describe the system independent facilities provided in *OS.Path*, whose signature is given in Figure 4.21. Some definitions are in order: an arc is a directory or a file relative to its own directory. Arc separator characters are used to separate arcs in paths. The character *#"/"* is the arc separator in Unix, *#"\\"* in DOS. The special arcs *parentArc* and *currentArc* correspond in Unix and DOS to *".."* and *"."* respectively. A path is a list of arcs, with an optional root. An absolute path has a root, while a relative path does not. A canonical path contains no occurence of either of the empty arc, the current arc (unless it is the only arc in the path) and can contain parent arcs only at the beginning and only if the path is relative. Intuitively, canonical paths do not contain useless arcs. Finally, paths have an associated volume, which is *""* under Unix and *"A:"*, *"C:"* and so on under DOS.

We will skip a lot of details, and address the important functions in this structure. More detail can as usual be found in the official Basis documentation. The basic functionality is given by *fromString* and *toString*, which respectively convert a string into and from a record {*isAbs,vol,arcs*} representing the path: *isAbs* is true if the path is absolute, *vol* is the volume while *arcs* is the list of arcs. The functions *getVolume*, and *getParent* respectively return the volume of a path and the parent directory of an entity pointed to by the path.

At the level of directories and files, the function *splitDirFile* splits a path into a directory part and a filename, while *joinDirFile* does the converse. The functions *dir* and *file* access the appropriate portions of *splitDirFile*.

In much a similar fashion, *splitBaseExt* splits a path into a base part and an extension part, while *joinBaseExt* does the converse and *base* and *ext* access the appropriate portions of *splitBaseExt*.

The predicates *isCanonical*, *isRelative*, *isAbsolute* and *isRoot* check the given path for the specified property.

Using *mkCanonical* one can convert a path to canonical form, while *mkAbsolute (p,abs)* will append the absolute path *abs* to the relative path *p*, unless *p* is already absolute. Conversely, *mkRelative (p,abs)* returns a path which when taken relative to the absolute path *abs* is equivalent to *p*. Finally, the function *concat* concatenates two paths and requires the second path to be relative. In all these functions, it is an error to consider paths on different volumes.

The substructure *OS.Process* provides functions dealing with processes in a system-independent way. Its signature is given in Figure 4.22. The type *status* represents the status or result of a process's execution, and two special values of that type are defined: *success* and *failure*. To execute a command from the underlying operating system, one calls *system* with a string argument to be interpreted by the underlying operating system or shell. The function blocks until the invoked process terminates, and it returns the status of the process. The call *exit* exits the

```
structure Process : sig
  eqtype status
  val success : status
  val failure : status
  val system : string -> status
  val atExit : (unit -> unit) -> unit
  val exit : status -> 'a
  val terminate : status -> 'a
  val getEnv : string -> string option
end
```

Figure 4.22: The structure *OS.Process*

```
structure Unix : sig
  type proc
  type signal
  val executeInEnv : (string * string list * string list) -> proc
  val execute : (string * string list) -> proc
  val streamsOf : proc -> (TextIO.instream * TextIO.outstream)
  val reap : proc -> OS.Process.status
  val kill : (proc * signal) -> unix
end
```

Figure 4.23: The structure *Unix*

process currently executing the program (or the compiler if *exit* is invoked at the interactive loop), passing a status back to the operating system. Calls to *exit* will flush all open output streams, close all open files, and execute all the actions registered to execute at exit-time. To register such an action, one calls *atExit*, passing it a *unit*→*unit* function. Exceptions raised by that function will be trapped and ignored when it invoked during an exit. To exit the process without triggering the *atExit* actions or flushing buffers, one may use the *terminate* function. Finally, the function *getEnv* looks up the value of the given environment variable as an option value, returning *NONE* if it cannot be found.

Other process-related functions can be found in the *Unix* structure, whose signature is given in Figure 4.23. Although it is inspired by Unix functionality, its functions should be useful in non-Unix settings.[3] The main function in *Unix* is *execute*, which takes a command and a list of arguments and asks the underlying operating system to execute the command as a separate process. Note that under Unix at least, the call is not routed through the shell, so there is no search performed, and the full path to the command is required. The result of the call is

[3]Unfortunately, SML/NJ currently only provides a structure *Unix* on Unix systems.

```
structure CommandLine : sig
  val name : unit -> string
  val arguments : unit -> string
end
```

Figure 4.24: The structure *CommandLine*

a value of type *proc* used to refer to that running process. Using *streamsOf*, one can obtain a pair of input and output streams hooked into the standard input and output of the process referred to by *proc*. Using *reap* will cause a suspension of the current process until the system process corresponding to *proc* has terminated, yielding its status. The semantics of Unix require that processes that have terminated be reaped, so a programming using *execute* should eventually invoke *reap* on any process it creates.

Finally, the structure *CommandLine*, whose signature is given in Figure 4.24, provides two functions for getting at the command used to invoke the currently running program (which is typically the interactive compiler itsefl). The function *name* returns the name of the command, while *arguments* returns a list of the arguments. Note that some of the command line arguments may been "consumed" by the runtime system, and may not appear in the list. For SML/NJ for example, command line arguments of the form @*SML*... prefix runtime system directives (which runtime system to use, which heap image, initial cache size, etc), which never filter to the executed program.

4.7 Time and dates

The structure *Time* provides an abstract notion of time and operations for manipulating it. Its signature is given in Figure 4.25. Time can be used in two distinct ways, and the library supports them both: time can be an absolute measure, as in the notion of "the current time", or time can be an interval, as in "how much time has elapsed since...". The distinction is artificial however, as absolute time can be seen as a time interval starting from some fixed point arbitrarily denoted time 0. Functions in the *Date* structure, which we will see later in this section, can convert this notion of absolute time with respect to some time 0 into an actual date and time. Given this duality, we shall frame our discussion using the interpretation of time as intervals.

Time is represent by an abstract type *time*. The value *zeroTime* denotes the empty time interval (and thus is the common reference point for specifying absolute

```
structure Time : sig
  eqtype time
  exception Time
  val zeroTime : time
  val fromReal : LargeReal.real -> time
  val toReal : time -> LargeReal.real
  val toSeconds : time -> LargeInt.int
  val toMilliseconds : time -> LargeInt.int
  val toMicroseconds : time -> LargeInt.int
  val fromSeconds : LargeInt.int -> time
  val fromMilliseconds : LargeInt.int -> time
  val fromMicroseconds : LargeInt.int -> time
  val + : (time * time) -> time
  val - : (time * time) -> time
  val compare : (time * time) -> order
  val < : (time * time) -> bool
  val <= : (time * time) -> bool
  val > : (time * time) -> bool
  val >= : (time * time) -> bool
  val now : unit -> time
  val fmt : int -> time -> string
  val toString : time -> string
  val fromString : string -> time option
  val scan : (char, 'a) StringCvt.reader -> 'a -> (time * 'a) option
end
```

Figure 4.25: The structure *Time*

time). The function *now* returns the "current time", that is the time interval since the fixed zero time point indicated by *zeroTime*. Further functions for creating and reading time values include *toReal* and *fromReal*, which convert a fractional number of seconds into and from a time value, while *toSeconds*, *toMilliseconds*, *toMicroseconds* and *fromSeconds*, *fromMilliseconds*, *fromMicroseconds* convert a number of seconds (respectively, milliseconds or microseconds) into and from a time value.

Arithmetic operations on time intervals are provided: + denotes a time interval which is the sum of the durations of its arguments, while - denotes a time interval which is the difference of its arguments. Since time values are required to be positive, *t1* must be less than *t2* in *- (t1,t2)*. A *Time* exception is raised otherwise. Note that *time* is an equality type, and moreover provides the standard comparison operators *compare*, $<$,$<=$,$>$ and $>=$.

Conversion to and from strings follow the usual pattern. The function *fmt* converts a time value to a string representing the time interval in seconds, with as argument an integer denoting the number of decimal digits to keep in the fractional part. Conversely, *scan* is a standard scanning function, turning a character reader into a time value reader (see Section 4.3). The time value scanned is taken as a (possibly fractional) number of seconds, as in the *fromReal* call. The function *toString* is equivalent to *fmt 3* and *fromString* is derived from *scan* in the usual way (it is equivalent to *StringCvt.scanString scan*).

Turning time values into actual dates and times is done via the *Date* structure, whose signature is given in Figure 4.26. Dates are represented through an abstract type which for all intents and purposes should be though of as a record {*year,month,day,hour,minute,second,offset*} where *year*, *month* and *day* give the date (*month* takes its values from a datatype *month* with values *Jan*,*Feb*,...), while *hour*,*minute* and *second* provide the time and *offset* is an optional value that indicates time zone information: a value of *NONE* indicates that the time is taken in the current time zone, while a value of *SOME (t)* corresponds to a time zone at time *t* west of UTC (Universal Time Coordinates, also known as Greenwich Mean Time or GMT).

The function *date* takes a record in the above form and converts it to a *date* value, possibily canonicalizing it (hours in excess of 24 carry over into days, and so on). The functions *year*, *month*, *day*, *hour*, *minute*, *second* and *offset* return the corresponding fields of the abstract time value — again, because of canonicalization the values retrieved may be different from what was passed in. The additional functions *weekday* return the day of the week corresponding to the date (given as a value from the datatype *weekday*), while *yearDay* returns the day of the year the date represents, with January 1 being day 0, February 1 being day 31, and so on. The function *isDst* returns *NONE* if the system has no information about daylight

```
structure Date : sig
  datatype weekday
    = Mon | Tue | Wed| Thu| Fri | Sat | Sun
  datatype month
    = Jan| Feb| Mar| Apr| May| Jun| Jul
    | Aug | Sep| Oct| Nov| Dec
  type date
  exception Date
  val date : {year : int, month : month, day : int, hour : int, minute : int,
    second : int, offset : Time.time option} -> date
  val year : date -> int
  val month : date -> month
  val day : date -> int
  val hour : date -> int
  val minute : date -> int
  val second : date -> int
  val weekDay : date -> weekday
  val yearDay : date -> int
  val offset : date -> Time.time option
  val isDst : date -> bool option
  val localOffset : unit -> Time.time
  val fromTimeLocal : Time.time -> date
  val fromTimeUniv : Time.time -> date
  val toTime : date -> Time.time
  val toString : date -> string
  val fmt : string -> date -> string
  val fromString : string -> date option
  val scan : (char, 'a) StringCvt.reader -> 'a -> (date * 'a) option
  val compare : (date * date) -> order
end
```

Figure 4.26: The structure *Date*

```
exists
 %% the percent character
 %c the character c, if c is not one of the format characters listed above
```

Figure 4.27: Formatting characters for *Date.fmt*

savings time (DST), *SOME (true)* if DST is in effect, and *SOME (false)* if it is not.

Functions for converting to and from time values (as defined in the *Time* structure) are provided. The function *localOffset* returns the offset corresponding to the current time zone (this is typically taken from the underlying operating system). Functions *fromTimeLocal* takes a time value and converts it to a date in the local time zone, interpreting the time value as the time interval since the time *Time.zeroTime*, while *fromTimeUniv* converts it to a date in the UTC time zone. Presumably,*fromTimeLocal* returns a date with an offset of *NONE* while *fromTimeUniv* returns a date with an offset of *SOME (0)*. The call *fromTimeLocal (Time.now ())* returns the current date and time, while *fromTimeLocal (Time.zeroTime)* returns the date corresponding to the common point for absolute time. Conversely, the function *toTime* converts a date value to a time value, that is the time elapsed from *Time.zeroTime* to the corresponding date, raising the *Date* exception if the date cannot be represented as a time value.

Comparison of dates is done through the standard *compare* function, while conversion to and from strings is done via functions *fmt* and *scan*. The function *fmt* takes a format string and a date and converts the date to a string according to the format. Format characters allowed in the format string are given in Figure 4.27. The function *toString* is an abbreviation fo *fmt*, hardwired to convert dates into strings of the form *"Wed Mar 08 19:06:45 1995"* with exactly 24 characters. The function *scan* is a standard scanning function, converting a character reader into a date value reader, where the date is parsed exactly as the format produced by *toString* (including possibly some initial whitespace). The function *fromString* is equivalent to *StringCvt.scanString scan*, as usual.

The *Date* structure is reasonably straightforward to use, although the handling of time zones takes some time to get used to. Mostly, this is due to people often counting offset from UTC in the range -11 to +11, and not 0 to -23, as is being done here. Two easy ways of getting out of this are possible. First, one can define a simple function from standard time offset to the Basis time offset and back, and second one may define standard time zones abbreviations for oft-used time offsets. Let's do both. Consider a structure *TimeZone* with functions *fromStdOffset* and *toStdOffset* to convert standard offsets to the Basis time offsets and back, and

```
structure TimeZone = struct
  exception TimeZone

  fun fromStdOffset (NONE) = NONE
    | fromStdOffset (SOME (v)) = SOME (if v > 0 andalso v <= 12
                                         then 24 - v
                                       else if (v <= 0 andalso v >= 12)
                                         then ~ v
                                       else raise TimeZone)

  fun toStdOffset (NONE) = NONE
    | toStdOffset (SOME (v)) = SOME (if v > 12
                                       then 24 - v
                                     else ~ v)   (* raise TimeZone? *)

  val EST = SOME (...)
  val GMT = SOME (...)
  val ...
end
```

Figure 4.28: The structure *TimeZone*

abbreviations such as *EST* for Eastern Standard Time, and so on. The structure is given in Figure 4.28.

The final structure from the Basis we consider is the *Timer* structure (signature in Figure 4.29), which provides functions to measure the passage of real time. Two types of timers are defined: real timers and cpu timers. A real timer keeps track of real time, while a cpu timer keeps track of time spent by the currently running process, that is the user time (time the process has had the CPU), the system time (time the process has been active in the kernel) and the GC time (time the process has spent on garbage collection). Once a timer is created, it starts keeping track of time, and can be queried for its current time count.

The functions *startRealTimer* and *startCPUTimer* return real and cpu timers respectively, which start keeping track of time, starting from time 0. The functions *checkRealTimer* and *checkCPUTimer* query the provided timer and return the time it is keeping track of. The functions *totalRealTimer* and *totalCPUTimer* return special timers, which keep track of elapsed time since a system-dependent initialization (typically, the start of the process itself).

As an example, consider the following function *timedFn*:

```
fun timedFn (f) a = let
  val ct = Timer.startRealTimer ()
  val result = f a
in
  (result,Timer.checkRealTimer (ct))
end
```

```
structure Timer : sig
  type cpu_timer
  type real_timer
  val startCPUTimer : unit -> cpu_timer
  val checkCPUTimer : cpu_timer -> {usr : Time.time, sys : Time.time, gc :
    Time.time}
  val totalCPUTimer : unit -> cpu_timer
  val startRealTimer : unit -> real_timer
  val checkRealTimer : real_timer -> Time.time
  val totalRealTimer : unit -> real_timer
end
```

Figure 4.29: The structure *Timer*

which takes a function of type *'a →'b* and creates a new function of type *'a →('b ×Time.time)*, which behaves like the original function, but moreover returns the time it took for the function to execute. We used real time in the example, but it is trivial to modify the function to keep track of cpu time.

4.8 Compatibility with SML'90

A final structure from the Basis that we cover is useful when porting programs written for SML'90. The *SML90* structure (whose signature is given in Figure 4.30) provides the bindings for types and values that were available at top-level with older versions of SML. The SML'90 version of the language defined a minimal library nowhere near the complexity of the current Basis. The *SML90* structure defines the types and values that were available in the old basis but that do not exist in the current one. For example, the list function *map* was available in the old basis, but is still present in the new Basis, and so is not implemented in *SML90*. Some differences between previous versions of SML and the SML'97 update involve semantic changes that cannot be isolated in the *SML90* structure. One example is that the *real* type was an equality type in previous versions of SML, while it is not so in SML'97. Similarly, arithmetic operations would raise specific exceptions in previous versions of SML, and this behavior cannot be captured in the *SML90* structure. Older programs relying on such features will need to be modified to take the new semantics into account

The *SML90* structure can be divided into three parts: math functions, string functions and I/O functions. The structure also defines exceptions, most of which are aliases for existing exception from elsewhere in the Basis. The math functions in *SML90* include *sqrt*, *exp*, *ln*, *sin*, *cos*, *arctan*, and most of these are equivalent to functions in the *Math* structure of the Basis. The string functions *ord*, *chr*, *explode*

```
structure SML90 : sig
     type instream
     type outstream
     exception Abs
     exception Quot
     exception Prod
     exception Neg
     exception Sum
     exception Diff
     exception Floor
     exception Exp
     exception Sqrt
     exception Ln
     exception Ord
     exception Mod
     exception Io of string
     exception Interrupt
     val sqrt : real -> real
     val exp : real -> real
     val ln : real -> real
     val sin : real -> real
     val cos : real -> real
     val arctan : real -> real
     val ord : string -> int
     val chr : int -> string
     val explode : string -> string list
     val implode : string list -> string
     val std_in : instream
     val open_in : string -> instream
     val input : (instream * int) -> string
     val lookahead : instream -> string
     val close_in : instream -> unit
     val end_of_stream : instream -> bool
     val std_out : outstream
     val open_out : string -> outstream
     val output : (outstream * string) -> unit
     val close_out : outstream -> unit
end
```

Figure 4.30: The structure *SML90*

and *implode* are also equivalent to functions in the Basis. Note that SML'90 did not have a specific type for characters. Typically, strings with a single character were used as a representation for that character. Hence, the type of *explode* in *SML90* is

```
val explode : string -> string list
```

while the *explode* function in the Basis structure *String* has type:

```
val explode : string -> Char.char list
```

The most extensive part of the *SML90* structure is the input/output facilities. It provides the basic facilities for opening and closing files, along with the input and output of character strings. The input and output is stream-based and buffered. As we shall see in Section 4.5 (and in more detail in Chapter **??**), the Basis provides a much more extensive subsystem, able to deal both with imperative and with stream-based I/O.

The types *instream* and *outstream* are the abstract types corresponding to stream-s of input and output created from files. The streams *std_in* and *std_out* are created when the compiler is first started and are bound to the standard input and output streams of the process that launched the compiler. For input, the function *open_in* opens a file whose name is passed as an argument and returns an *instream* on the given file; *input (s,n)* returns a string of at most *n* characters read from the supplied input stream *s*; *lookahead* returns the next character in the stream, or an empty string if the end of the file has been reached; *end_of_stream* returns true if the end of the file has been reached. Note that all these operations are blocking: if the end of the file has not been reached, but not all the required characters are available, the call blocks until the characters become available. The function *close_in* closes the given input stream. Corresponding functions exist for output: *open_out* returns an *outstream* for output to the given file; *output* sends a given string to the given output stream; *close_out* closes the output stream.

The exception *Io* is used to indicate errors, such as trying to read from or write to a close stream, or trying to open a file for input and failing (say because the file does not exist), or trying to open a file for output and failing (say because the user does not have write permissions to the directory in which the file is to be created).

Notes

The official documentation to the basis will be published in [35] and a draft version is available online from the SML/NJ web site. Most of the discussion in these notes summarize the descriptions in the official documentation. Gilmore's notes

[37] provide a nice overview of some relevant aspects of the Basis Library in his tutorial notes for programming in SML'97.

Some of the issues about conventions in the Basis could be automatically enforced by a more powerful type system, such as one based on type classes [112].

In pre-SML'97, real numbers actually formed an equality type. The current implementation of reals is based on the IEEE standards [53] and [54] and the notes [55]. The *Math* structure is based on its C library <*math.h*> counterpart. .

Some structures were left out of the discussion, namely *PackReal* and *Byte*. These described in the official documentation. They are mostly useful for with low-level programming. Note that the release version of SML/NJ does not implement *PackReal*.

The use of folding functions to implement list-based processing functions (and in general, other iterative structures) is well-known in the functional programming community, and has lead to a whole theory of such functions. Refer, for example, to the work of Meijer et al. [70].

The *OS.IO* substructure implements polling as per the Unix SVR4 interface.

The best reference for such notions as canonical dates, and deriving weekdays from dates, and converting from time to date and back is the book by Dershowitz and Reingold [26].

A great number of structure we have not discussed are implemented on Posix systems, dealing exclusively with providing access to Posix functionality. Such functionality is used for example to build the IO infrastructure on Posix systems, and in fact is the only way to access the terminal to do cursor addressing.

The IO subsystem is described in much more detail in Chapter **??**. Sockets are described independently in Chapter **??**.

Part II

Standard ML of New Jersey

Chapter 5

The Interactive Compiler

What we have discussed until now in these notes is Standard ML, the language, along with its extensive Basis Library. In this chapter and the ones following, we begin an in-depth description of Standard ML of New Jersey, an implementation of that language, a compiler and an environment. We explore the aspects of that environment, interacting with the compiler and the support tools, as well as the SML/NJ library, which complements the Basis Library in its support for data structures and utility functions.

In this chapter, the focus is on the interaction with the compiler. The compiler runs an interactive loop, waiting for the user to enter code, which the system compiles and executes. We saw in Section 1.6 the basics of starting and terminating the interactive compiler, using the interactive loop, and reading error messages and such. Here, we focus on more in-depth issues such as controlling and customizing details of the system and the compiler, accessing aspects of the compiler such as the evaluation stream and the prettyprinter, as well as managing heap images and generating standalone programs.

5.1 Controlling the runtime system

In this section, we look at the facilities for tweaking the settings in the system. It provides a deeper look at the *SMLofNJ* structure, which defines facilities specific to the runtime system of SML/NJ. We will return to some of the functionality in this structure later in this chapter, as well as in later chapters (specifically, Chapter **??**). The signature of *SMLofNJ* is given in Figure 5.1.

We already saw a description of most of the functions in *SMLofNJ*. The function *exnHistory* returns, for a given raised exception, a list of strings describing where the exception was raised in the code, and through which handler it has

```
structure SMLofNJ : sig
  structure Cont : CONT
  structure IntervalTimer : INTERVAL_TIMER
  structure Internals : INTERNALS
  structure SysInfo : SYS_INFO
  structure Susp : SUSP
  structure Weak : WEAK
  val exportML : string -> bool
  val exportFn : (string * ((string * string list) ->
                              OS.Process.status))
                 -> unit
  val getCmdName : unit -> string
  val getArgs : unit -> string list
  val getAllArgs : unit -> string list
  datatype 'a frag
  = QUOTE of string
  | ANTIQUOTE of 'a
  val exnHistory : exn -> string list
end
```

Figure 5.1: The structure *SMLofNJ*

passed. The *frag* datatype will be discussed in Chapter **??** along with the *Susp* and *Weak* substructures, while the *Cont* substructure is the subject of Chapter **??**. We focus here on the remaining substructures.

The structure *SMLofNJ.SysInfo* provides information on the current system, that is the system hosting the currently executing runtime system. Its signature is given in Figure 5.2. Its main functions are *getOSKind* which returns an element of type *os_kind* describing the operating system (*UNIX*, *WIN32*, *MACOS*, ...), while *getOSName* returns the actual name of the operating system and *getOSVersion* its version. The functions *getHostArch* and *getTargetArch* return a string respectively describing the architecture of the system running the compiler and the architecture for which the compiler is generating code. Unless one is cross-compiling, these values are the same. Cross-compiling is interesting, but beyond the scope of these notes.

The structure *SMLofNJ.IntervalTimer* (signature in Figure 5.3) allows one to set the granularity of the timers (for example, those created by the *Timer* structure of the Basis, see Section 4.7). The function *tick* returns the current granularity, that is the smallest interval of time that the timers can measure. The function *setIntTimer* changes this granularity to the specified time value, or disables timers altogether if *NONE* is given as an argument.

The substructure *SMLofNJ.Internals* provides access to the true core of the system. Its signature is given in Figure 5.4. The value *prHook* is a hook for the

```
structure SMLofNJ.SysInfo : sig
  exception UNKNOWN
  datatype os_kind
  = UNIX
  | WIN32
  | MACOS
  | OS2
  | BEOS
  val getOSKind : unit -> os_kind
  val getOSName : unit -> string
  val getOSVersion : unit -> string
  val getHostArch : unit -> string
  val getTargetArch : unit -> string
  val hasSoftwarePolling : unit -> bool
  val hasMultiprocessing : unit -> bool
end
```

Figure 5.2: The structure *SMLofNJ.SysInfo*

```
structure IntervalTimer : sig
  val tick : unit -> Time.time
  val setIntTimer : Time.time option -> unit
end
```

Figure 5.3: The structure *IntervalTimer*

```
structure SMLofNJ.Internals : sig
  structure CleanUp : CLEAN_UP
  structure GC : GC
  val prHook : (string -> unit) ref
  val initSigTbl : 'a -> unit
  val clearSigTbl : 'a -> unit
  val resetSigTbl : 'a -> unit
end
```

Figure 5.4: The structure *SMLofNJ.Internals*

```
structure CleanUp : sig
  datatype when= AtExportML| AtExportFn| AtExit| AtInit| AtInitFn
  val atAll : when list
  val addCleaner : (string * when list * (when -> unit)) -> (when list * (when ->
    unit)) option
  val removeCleaner : string -> (when list * (when -> unit)) option
  val clean : when -> unit
end
```

Figure 5.5: The structure *CleanUp*

top-level *print* function, allowing it to be rebound: in effect, *print* calls whatever function is stored in this reference cell (initially set to *TextIO.print*). Substructures of interest include *CleanUp* and *GC*.

The substructure *SMLofNJ.Internals.CleanUp* (signature in Figure 5.5) allows the user to control actions perform at certain key events in the life of a SML process, such as process exit or heap initialization. In Section 4.6, we already saw how to register actions to be performed at exit time. This structure completes the picture, by providing more events of interest to attach actions to. The datatype *when* denotes events of interest, with values *atExportFn*, *atExportML*, *atExit*, *atInit*, *atInitFn*, the last two corresponding to the event of starting to execute code in a loaded heap image generated by *exportML* and *exportFn* respectively (exporting heaps will be described in Section 5.4). The value *atAll* contains a list of all allowable events. A cleaner is a function invoked at some event: it is defined by a name (of type *string*), and an action to be performed when an event occurs (of type *event* →*unit*). The action is passed the event that triggered it. Typical actions include closing files that have been left open, and such. To add a new cleaner to the system, one calls *addCleaner* which takes a name for the cleaner, a list of events it is registered to react to, and the cleaner action proper. The function returns an option value, with any existing cleaner with the same name, if one exists. Such an already existing cleaner is removed when a new one is installed. To remove a cleaner without installing a new one, we can use *removeCleaner* passing in the cleaner name. To manually invoke the cleaners associated with an event, one can call *clean* passing in the event value to simulate. Of course, cleaners get invoked automatically when the associated event actually happens.

Although a cleaner gets removed when a cleaner of the same name is installed, using higher-order functions, one can easily combine cleaners instead of removing old ones:

```
structure GC : sig
  val doGC : int -> unit
  val messages : bool -> unit
end
```

Figure 5.6: The structure *GC*

```
fun combineCleaner (name,when,f) =
  case SMLofNJ.Internals.CleanUp.addCleaner (name,when,f)
    of NONE => ()
     | SOME (name',when',f') => SMLofNJ.Internals.CleanUp.addCleaner....
           (* idea: add new cleaner of the same name, with a combined
action... *)
(* let w = when' - when in ... addClearner (n,w,f) then combine... *)
```

This is not an especially recommended approach, but shows once again how higher-order functions can be very useful.

The substructure *SMLofNJ.Internals.GC* provides a very simple interface to the garbage collector. The signature is given in Figure 5.6. The function *doGC* invokes the garbage collector manually, passing in the highest generation to consider while collecting (SML/NJ uses a generational garbage collector). In effect, *doGC (0)* performs a quick garbage collection of the allocation space, while *doGC (1000)* performs a major collection, scavenging all garbage. The more thorough the collection, the longer garbage collection will take. The function *messages* enables or disables garbage collection messages. By default, messages are enabled for heaps created by *exportML* and disabled for heaps created by *exportFn* — thus interactive systems show garbage collection messages (as SML/NJ itself does), but not standalone applications.

5.2 Controlling the compiler

In the previous section, we have seen the customization facilities provided by *SMLofNJ* for the system in its generality, that is facilties that affect every heap image. In this section and the next ones, we focus on the compiler itself.

The SML/NJ compiler is accessible through the top-level structure *Compiler* (at least, the so-called visible compiler is). This allows user programs, such as CM, the possibility of manipulating the compiler. Figure 5.7 gives the signature for this structure, and we will focus on various specific parts of the structure, as most of it involves the compilation process itself which, though fascinating in its own right, is beyond the scope of these notes. The substructures of interest to us are *Control*, *PrettyPrint*, *PPTable* and *Interact*.

```
structure Compiler : sig
  structure Stats : STATS
  structure Control : CONTROL
  structure Source : SOURCE
  structure SourceMap : SOURCE_MAP
  structure ErrorMsg : ERRORMSG
  structure Symbol : SYMBOL
  structure StaticEnv : STATICENV
  structure DynamicEnv : DYNENV
  structure BareEnvironment : ENVIRONMENT
  structure Environment : ENVIRONMENT
  structure CoerceEnv : COERCE_ENV
  structure EnvRef : ENVREF
  structure ModuleId : MODULE_ID
  structure SCStaticEnv : SCSTATICENV
  structure Profile : PROFILE
  structure CUnitUtil : CUNITUTIL
  structure CMSA : CMSA
  structure PersStamps : PERSSTAMPS
  structure PrettyPrint : PRETTYPRINT
  structure PPTable : sig
  val install_pp : string list -> (PrettyPrint.ppstream -> 'a -> unit) ->
  unit
  end
  structure Ast : AST
  structure Lambda : sig
  type lexp
  end
  structure Compile : COMPILE
  structure Interact : INTERACT
  structure Machm : CODEGENERATOR
  structure PrintHooks : PRINTHOOKS
  structure Boot : sig
  val coreEnvRef : SCEnv.Env.environment ref
  end
  val version : system : string, version_id : int list, date : string
  val banner : string
  val architecture : string
end
```

Figure 5.7: The structure *Compiler*

```
structure Compiler.Control : sig
  structure MC : MCCONTROL
  structure Lazy : LAZYCONTROL
  structure CG : CGCONTROL
  structure Print : PRINTCONTROL
  val debugging : bool ref
  val primaryPrompt : string ref
  val secondaryPrompt : string ref
  val printWarnings : bool ref
  val valueRestrictionWarn : bool ref
  val instantiateSigs : bool ref
  val internals : bool ref
  val interp : bool ref
  val saveLambda : bool ref
  val saveLvarNames : bool ref
  val preserveLvarNames : bool ref
  val markabsyn : bool ref
  val trackExn : bool ref
  val indexing : bool ref
  val instSigs : bool ref
  val quotation : bool ref
  val saveit : bool ref
  val saveAbsyn : bool ref
  val saveConvert : bool ref
  val saveCPSopt : bool ref
  val saveClosure : bool ref
  val lambdaSplitEnable : bool ref
  val crossInlineEnable : bool ref
end
```

Figure 5.8: The structure *Control*

First, three values in *Compiler* are useful. The string *banner* is displayed by the default heap on startup, giving the name and version of the compiler. The value *version* contains a non-string version of this information (more appropriate if say your program needs to check the current version of the compiler that you are using) while *architecture* is a short identifier for the instruction-set architecture on which the system is running.

The *Compiler.Control* substructure provides convenient flags to control various aspects of the compiler. Its signature is given in Figure 5.8. Flags are typically given as boolean references, so that an assignment to the appropriate flag will affect every subsequent compilation. The flags of interest for us in this structure are the string references *primaryPrompt* and *secondaryPrompt* which hold the strings representing the prompts to be displayed to the user in the interactive loop (*"-"* and *"="*, respectively). The secondary prompt is displayed when the input continues past the first line entered. The boolean flags *printWarnings* controls whether warn-

```
structure Compiler.Control.Print : sig
  val printDepth : int ref
  val printLength : int ref
  val stringDepth : int ref
  val printLoop : bool ref
  val signatures : int ref
  val printOpens : bool ref
  val out : say : string -> unit, flush : unit -> unit ref
  val linewidth : int ref
  val say : string -> unit
  val flush : unit -> unit
end
```

Figure 5.9: The structure *Print*

ing messages from the compiler are displayed. Warnings typically indicate that although an error has not occurred, some corrective or non-obvious default action has been taken that the user may not be aware of.

A special case of this, the boolean flag *valueRestrictionWarn*, controls whether the system displays a warning when the compiler fails to generalize a polymorphic value inside a *let* binding. The typical corrective action taken is to instantiate the polymorphic type to a new type not used anywhere else, which means of course that the polymorphic value is unusable (it will fail to type-check if used anywhere), but does allows the code itself to compile. . We will return to the boolean flag *quotation* in Chapter **??** — this flag controls the use of the backquote as a special quotation character.

The *Compiler.Control* structure also defines further substructures controlling more specialized aspects of the compiler. We focus now on the printing mechanism in substructure *Print*. The signature is given in Figure 5.9. It also provides a number of reference cells that can be set to different values. The values *printDepth*, *printLength* and *stringDepth* control at which point the display of values by the compiler is truncated and ellipses (...) are printed. The flag *printDepth* controls how many levels of nesting are displayed for recursive data structures (user-defined, not including lists), the flag *printLength* controls how many elements of a list are displayed, and the flag *stringDepth* controls how many characters of a string are displayed. The flag *printLoop* specifies whether to treat loops (involving reference cells) specially when printing. For example, consider the following code, which defines a type and a value of that type.

```
- datatype node = Node of int * node option ref;
datatype node = Node of int * node option ref
- val v = Node (0,ref NONE);
val v = Node (0,ref NONE) : node
```

Next, we create a loop by having the reference cell in *v* point to *v* itself. The following code does the job:

```
- let val Node (_,r) = v in r := SOME (v) end;
val it = () : unit
```

Let us now see how the value *v* gets printed at the toplevel. First, using the default value for *printLoop*, which is *true* (note that we set the print depth to a suitable value to get non-trivial output):

```
- Compiler.Control.Print.printDepth := 100;
val it = () : unit
- v;
val it = Node (0,ref (SOME (Node (0,%0))) as %0) : node
```

We see that the compiler has detected a loop in the output, and use the placeholder *%0* to point out the loop. If *printLoop* is set to *false*, things are not so nice:

```
- Compiler.Control.Print.printLoop := false;
val it = () : unit
- v;
val it =
  Node
    (0,
     ref
       (SOME
          (Node
             (0,
              ref
                (SOME
                   (Node
                      (0,
                       ref
                         (SOME
                            (Node
...
```

and the output goes on for quite some time (until the printing depth has been reached).

The flags *signatures* and *printOpens* control the printing of signature bodies respectively when a structure with that signature is opened. The flag *linewidth* specifies how many characters should fit in a line for the purpose of prettyprinting values (see Section 5.3 later in this chapter). The flag *out* specifies how the printing of compiler messages should be performed. The value stored is a record $\{$*say : string* $\rightarrow$*unit, flush : unit* $\rightarrow$*unit*$\}$. The field *say* specifies how to print to top-level, and the field *flush* how to flush the buffer associated with printing to top-level, if applicable. By default, printing is done on standard output, so the corresponding would be

```
{say = fn (s) => TextIO.output (TextIO.stdOut,s),
 flush = fn () => TextIO.flushOut (TextIO.stdOut)}
```

On the other hand, the following suppresses all output from the compiler:

```
structure Compiler.Interact : sig
  exception Interrupt
  val interact : unit -> unit
  val useFile : string -> unit
  val useStream : TextIO.instream -> unit
end
```

Figure 5.10: The structure *Interact*

```
{say = fn _ => (), flush = fn () => ()}
```

One can write a function *useSilently* to *use* a file but without displaying any output associated with the loading (aside for explicit printing in the file), as in:

```
fun useSilently (s) = let
  val saved = !Compiler.Control.Print.out
  fun done () = Compiler.Control.Print.out := saved
in
  Compiler.Control.Print.out := {say = fn _ => (), flush = fn () => ()}
  (use (s); done ()) handle _ => done ()
end
```

Note that we are cheating: we noted in Section 1.6 that *use* should only be used at top-level. Here we are using it within the body of a function. It turns out to work in this case, but future versions of the compiler may change this behavior, breaking the above code.

The functions *say* and *flush* in *Compiler.Control.Print* simply call the functions stored in the corresponding fields of *out*, as a shortcut. User will typically not use these functions, unless they want their output to be considered compiler-related and controlled by this compiler flag.

The last substructure of *Compiler* we discuss in this section, not formally concerned with customizing the compiler, is *Compiler.Interact*, which among others includes facilities for compiling code other than what is entered at the top-level loop. Part of the signature of *Compiler.Interact* is given in Figure 5.10. The function *interact* launches a new interactive loop, while *useFile* is the underlying binding of the toplevel function *use*. The function *useStream* is a generalization, compiling code from a specified imperative stream. In fact, we can define:

```
val useFile (s) = let
  val ins = TextIO.openIn (s)
in
  Compiler.Interact.useStream (ins);
  TextIO.closeIn (ins)
end
```

With *useStream*, we can write functions such as

```
structure Compiler.PrettyPrint : sig
  type ppstream
  type ppconsumer
  datatype break_style= CONSISTENT| INCONSISTENT
  exception PP_FAIL of string
  val mk_ppstream : ppconsumer -> ppstream
  val dest_ppstream : ppstream -> ppconsumer
  val add_break : ppstream -> (int * int) -> unit
  val add_newline : ppstream -> unit
  val add_string : ppstream -> string -> unit
  val begin_block : ppstream -> break_style -> int -> unit
  val end_block : ppstream -> unit
  val clear_ppstream : ppstream -> unit
  val flush_ppstream : ppstream -> unit
  val with_pp : ppconsumer -> (ppstream -> unit) -> unit
  val pp_to_string : int -> (ppstream -> 'a -> unit) -> 'a -> string
end
```

Figure 5.11: The structure *Compiler.PrettyPrint*

```
fun useString (s) = let
  val ins = TextIO.openString (s)
in
  Compiler.Interact.useStream (ins);
  TextIO.closeIn (ins)
end
```

allowing for some dynamic control over what can be defined at top-level. Notice however that the mechanism for *use* corresponds to: compile and execute the code. The result of computing the expression, if any, gets printed to the default output stream and *useStream* (and *useFile*) return *()*. There is no way to directly obtain the result of the compilation, short of redirecting the output of the compiler (via say *Compiler.Control.Print.out*) and parsing the result by hand. For example, evaluating

```
- useString "3+4;";
val it = 7 : int
val it = () : unit
```

yields the final value *()*, while the intermediate steps output the result of the compiled code. There is no type-safe way to execute dynamically generated code and get a meaningful value back. As an example of the kind of problem that could occur, what would such an eval function return for values of a user-defined type?

5.3 Prettyprinting

The *Compiler.PrettyPrint* structure, whose signature is given in Figure 5.11, implements functions to define prettyprinters for monomorphic user-defined types,

```
structure SimpleXml : sig
  datatype simple_xml = Word of string
                      | Tagged of {tag : string,
                                   contents: simple_xml list}

  val listOfWords : string -> simple_xml list
  val toString : simple_xml -> string
end = struct
  datatype simple_xml = Word of string
                      | Tagged of {tag : string,
                                   contents: simple_xml list}

  fun listOfWords (s:string): simple_xml list =
    map Word (String.tokens Char.isSpace s)

  fun toString (e:simple_xml):string = let
    val c = String.concat
  in
    case e
      of Word (s) => s^" "
       | Tagged {tag,contents} => c ["<",tag,">",
                                     c (map toString contents),
                                     "</",tag,">"]
  end
end
```

Figure 5.12: The structure *SimpleXML*

which is used to print values of those types in the top-level interactive loop. Generally, a prettyprinter takes a stream of characters and prints them in an aesthetically pleasing way, with appropriate indentation and line breaks.

Most examples of prettyprinting are taken from the prettyprinting of programming languages. However, for the sake of simplicity, we illustrate prettyprinting ona small but hopefully illustrative example. Consider the problem of representing marked-up text, in a way reminiscent of XML. The structure *SimpleXML* in Figure 5.12 gives the implementation I have in mind. The structure defines a datatype *simple_xml* to represent marked-up text, and utility functions to operate on values of that type. A value of type *simple_xml* is either a word or a tagged element, consisting of a tag and a list of contained elements. The representation of the XML-like text

```
<SOME>This is text</SOME>
```

would be written as:

```
Tagged {tag="SOME",content=[Word "this", Word "is", Word "text"]}
```

```
local
  open SimpleXml
  fun speech (l:simple_xml list):simple_xml =
    Tagged {tag="SPEECH",contents=l}
  fun speaker (s:string):simple_xml =
    Tagged {tag="SPEAKER",contents=listOfWords (s)}
  fun line (s:string):simple_xml =
    Tagged {tag="LINE",contents=listOfWords (s)}
  fun stagedir (s:string):simple_xml =
    Tagged {tag="STAGEDIR",contents=listOfWords (s)}
  val speech1 = speech [speaker "First Clown",
                        line "A pestilence on him for a mad rogue! a’ poured a",
                        line "flagon of Rhenish on my head once. This same skull,",
                        line "sir, was Yorick’s skull, the king’s jester."]
  val speech2 = speech [speaker "HAMLET",
                        line "This?"]
  val speech3 = speech [speaker "First Clown",
                        line "E’en that."]
  val speech4 = speech [speaker "HAMLET",
                        line "Let me see.",
                        stagedir "Takes the skull",
                        line "Alas, poor Yorick! I knew him, Horatio: a fellow",
                        line "of infinite jest, of most excellent fancy: he hath",
                        line "borne me on his back a thousand times; and now, how"]
in
  val hamlet =
    Tagged {tag="EXTRACT",
            contents=[speech1,speech2,speech3,speech4]}
end
```

Figure 5.13: A passage from Shakespeare

and so on. The function *listOfWords* helps constructing such values by parsing a string into constituent words (note that it does not handle markup tags). Thus, the above can be produced by:

```
Tagged {tag="SOME",contents=listOfWords "this is text"}
```

The function *toString* converts a value of type *simple_xml* to a string representation in the example above.

Consider the representation of this famous passage from Shakespeare’s Hamlet, using a fictional tagging scheme for plays, given in Figure 5.13. Displaying the resulting value at the prompt does not produce the most readable output:

```
- hamlet;
val it =
  Tagged
    {contents=[Tagged
                 {contents=[Tagged
                              {contents=[Word "First",Word "Clown"],
                               tag="SPEAKER"},
                            Tagged
                              {contents=[Word "A",Word "pestilence",Word "on",
                                         Word "him",Word "for",Word "a",
                                         Word "mad",Word "rogue!",Word "a'",
                                         Word "poured",Word "a"],tag="LINE"},
                            Tagged
                              {contents=[Word "flagon",Word "of",
                                         Word "Rhenish",Word "on",Word "my",

                                   ...
                                   ...
                                         Word "thousand",Word "times;",
                                         Word "and",Word "now,",Word "how"],
                               tag="LINE"}],tag="SPEECH"}],tag="EXTRACT"}
  : SimpleXml.simple_xml
```

Notice that the top-level reports the value *hamlet* as a member of the *simple_exp* type. We can display *hamlet* in a readable form by calling, say,

```
- print (SimpleXml.toString (hamlet));
<EXTRACT><SPEECH><SPEAKER>First Clown </SPEAKER><LINE>A pestilence on
him for a mad rogue! a' poured a </LINE><LINE>flagon of Rhenish on my
head once. This same skull, </LINE><LINE>sir, was Yorick's skull, the
king's jester. </LINE></SPEECH><SPEECH><SPEAKER>HAMLET
</SPEAKER><LINE>This? </LINE></SPEECH><SPEECH><SPEAKER>First Clown
</SPEAKER><LINE>E'en that. </LINE></SPEECH><SPEECH><SPEAKER>HAMLET
</SPEAKER><LINE>Let me see. </LINE><STAGEDIR>Takes the skull
</STAGEDIR><LINE>Alas, poor Yorick! I knew him, Horatio: a fellow
</LINE><LINE>of infinite jest, of most excellent fancy: he hath
</LINE><LINE>borne me on his back a thousand times; and now, how
</LINE></SPEECH></EXTRACT>
```

But this is painful to do everytime, and moreover the result is not very readable! We will eventually solve both problems, but for now, let us attack one problem at a time: we will write a trivial prettyprinter (that does nothing but convert the output to a string as *toString* does), and tell the system about it, to get the interactive loop to automatically display the value in “readable” form. Then, we improve the prettyprinter to make the displayed *simple_xml* value more aesthetically pleasing.

Prettyprinters are defined to work on streams of tokens. A prettyprinter outputs to a *ppstream*, which is used by a *ppconsumer*. The standard *ppconsumer* is the interactive loop. Given a *ppconsumer*, the function *mk_ppstream* creates a *ppstream* to write to that consummer. The prettyprinting process consists of taking that *ppstream* and sending the strings to be written to it (one can recuperate the *ppconsumer* for a *ppstream* by calling *dest_ppstream*). Let us write our first (trivial) prettyprinter for *simple_xml* values. It takes a *ppstream* to send the output to, and a value to send to it (it is curried for reasons to be seen later):

```
fun ppXml1 ppstream v = let
  val s = SimpleXml.toString (v)
in
  Compiler.PrettyPrint.add_string ppstream s
end
```

It simply converts the value to a string and sends the string to the *ppstream*, without special formatting. As I said, this prettyprinter is trivial — it does not prettyprint at all. Nevertheless, we can still tell the interactive loop about it to display "readable" value. The trick is general, in that we can produce custom display methods for different types at the interactive loop, even if no actual pretty printing is desired.

The interactive loop maintains a table of prettyprinters for various types, and when asked to print a value, it refers to that table to see if a prettyprinter for the type of that value is available; if so, it uses it to print the value, otherwise it uses the default printer. Thus, we only need to add our *ppExp1* prettyprinter to the table to get *simple_xml* values displayed automatically. A function *Compiler.PPTable.install_pp* does this: it takes as input a list of strings representing a path to the type the prettyprinter is for (in our case, since we want a prettyprinter for *SimpleXml.simple_xml*, this is *["SimpleXml","simple_xml"]*), and the corresponding prettyprinter, of type *ppstream →'a →unit*, where *'a* should be the type pointed to by the path. So, let's try:

```
- Compiler.PPTable.install_pp ["SimpleXml","simple_xml"] ppXml1;
val it = () : unit
- hamlet;
val it =
  <EXTRACT><SPEECH><SPEAKER>First Clown </SPEAKER><LINE>A pestilence on him for a mad rogue! a' poure
  : SimpleXml.simple_xml
```

Of course, the output is still not very nice, but it is a step up from the datatype representation we had before (unless of course, one prefers the datatype representation... it has the advantage of being unambiguous).

As we mentionned, for any type for which a *toString* function is defined, we can automatically generate a trivial prettyprinter. This feature is useful, so let us lift it into a general function:

```
fun mk_pp (f : 'a -> string) ppstream v = let
  val s = f (v)
in
  Compiler.PrettyPrint.add_string ppstream s
end
```

Let us write a slightly improved prettyprinter. This time, we make sure that words and tags do not wrap around the end of the screen, but rather go on the next line if they do not fit at the end of a given line. As a first step, let us decouple the prettyprinter from *toString*, by directly printing every element to the stream.

```
fun ppXml1' ppstream v = let
  fun pr s = Compiler.PrettyPrint.add_string ppstream s
  fun pp (e:SimpleXml.simple_xml):unit =
    (case e
       of SimpleXml.Word (s) => (pr s;
                                pr " ")
        | SimpleXml.Tagged tag,contents => (pr "<";
                                            pr tag;
                                            pr ">";
                                            app pp contents;
                                            pr "</";
                                            pr tag;
                                            pr ">"))
in
  pp v
end
```

As you can try, this behaves as *ppXml1*, by sending each string independently. The system still does not handle wraparounds, because we did not tell it how! How is it usually handled anyways? Time to turn to how in general prettyprinting works.

We have seen the *add_string* primitive which adds a string to the *ppstream*. The algorithm for prettyprinting is not allowed to break a line within a string, as we saw in the above example. Delimiters are used to indicate to the prettyprinting algorithm where it can break lines. The first type of delimiter is the blank (which includes CR, LF, FF, and spaces, as defined by *Char.isSpace*). The prettyprinting algorithm is allowed to break lines at blanks. A call to *add_break* inserts a blank in the *ppstream*, with an argument *(i,j)* giving respectively the size and offset of the blank. The behavior is as follows: if the element following the blank fits on the line (taking size *i* into consideration), then a blank of size *i* is printed out. Otherwise, a newline is performed, and a number of spaces equal to the blank offset is output before proceeding. Let us rewrite *ppXml1'* replacing every space by a break of size 1 and offset 2 (so that when a line is broken, it gets offset by 2 before being printed out):

```
fun ppXml2 ppstream v = let
  fun pr s = Compiler.PrettyPrint.add_string ppstream s
  fun break () = Compiler.PrettyPrint.add_break ppstream (1,2)
  fun pp (e:SimpleXml.simple_xml):unit =
    (case e
       of SimpleXml.Word (s) => (pr s;
                                 break ())
        | SimpleXml.Tagged tag,contents => (pr "<";
                                            pr tag;
                                            pr ">";
                                            break ();
                                            app pp contents;
                                            pr "</";
                                            pr tag;
                                            pr ">";
                                            break ()))
in
  pp v
end
```

Install and test the prettyprinter as before:

```
- Compiler.PPTable.install_pp ["SimpleXml","simple_xml"] ppXml2;
val it = () : unit
- hamlet;
val it = <EXTRACT> <SPEECH> <SPEAKER> First Clown </SPEAKER> <LINE> A
    pestilence on him for a mad rogue! a' poured a </LINE> <LINE> flagon of
    Rhenish on my head once. This same skull, </LINE> <LINE> sir, was Yorick's
    skull, the king's jester. </LINE> </SPEECH> <SPEECH> <SPEAKER> HAMLET
    </SPEAKER> <LINE> This? </LINE> </SPEECH> <SPEECH> <SPEAKER> First Clown
    </SPEAKER> <LINE> E'en that. </LINE> </SPEECH> <SPEECH> <SPEAKER> HAMLET
    </SPEAKER> <LINE> Let me see. </LINE> <STAGEDIR> Takes the skull
    </STAGEDIR> <LINE> Alas, poor Yorick! I knew him, Horatio: a fellow </LINE>
    <LINE> of infinite jest, of most excellent fancy: he hath </LINE> <LINE>
    borne me on his back a thousand times; and now, how </LINE> </SPEECH>
    </EXTRACT>  : SimpleXml.simple_xml
```

This is already much better! But there is still more to be done to get a nice expression display. To get more control over the prettyprinted output, we introduce the notion of a logical block (or simply a block). Special delimiters {| and |} denote the starting and ending point of a logically contiguous block of elements. Fundamentally, the algorithm will try to break as few blocks as possible onto different lines.

A starting block delimiter is added to the *ppstream* by a call to *begin_block pp bs i*, where *pp* is again the *ppstream*, *bs* is the break style (*CONSISTENT* or *INCONSISTENT*) and *i* is the block offset. The end of the block is indicated by an *end_block* call. A block is always indented at least as much as the original point at which it appears in the output, and moreover each line after the first line of the block (if any) is offset by an extra *i* spaces (to which, in turns, the extra space offset specified by any *add_break* gets added). The break style specifies how the different elements of the block (break-separated) get output. A *CONSISTENT* break style means that if the blocks gets printed across multiple lines because it does not fit on one line, every break in the block is turned into a carriage return, printing each element of the block on a different line (each indented and offset according to block offset and break offset). An *INCONSISTENT* break style allows multiple block elements on any line.

Let us now adapt our *simple_xml* values prettyprinter to take blocks into account. Informally, every tagged element forms a logical block. We set the break style to *INCONSISTENT* and prescribe a block offset of 2. We also remove the break offsets. The break/block offset game is much more a matter of taste than anything else. We get:

```
fun ppExp3 ppstream v = let
  fun pr s = Compiler.PrettyPrint.add_string ppstream s
  fun break () = Compiler.PrettyPrint.add_break ppstream (1,0)
  fun begin () = Compiler.PrettyPrint.begin_block ppstream Compiler.PrettyPrint.INCONSISTENT
  fun stop () = Compiler.PrettyPrint.end_block ppstream
  fun pp (e:SimpleXml.simple_xml):unit =
    (case e
       of SimpleXml.Word (s) => (pr s;
                                 break ())
        | SimpleXml.Tagged tag,contents => (begin ();
                                            pr "<";
                                            pr tag;
                                            pr ">";
                                            break ();
                                            app pp contents;
                                            pr "</";
                                            pr tag;
                                            pr ">";
                                            stop ();
                                            break ()))
in
  pp v
end
```

Installing and testing this last prettyprinter for *simple_xml* values, we get the satisfying output:

```
val it = () : unit
- hamlet;
val it =
  <EXTRACT>
    <SPEECH> <SPEAKER> First Clown </SPEAKER>
      <LINE> A pestilence on him for a mad rogue! a' poured a </LINE>
      <LINE> flagon of Rhenish on my head once. This same skull, </LINE>
      <LINE> sir, was Yorick's skull, the king's jester. </LINE> </SPEECH>
    <SPEECH> <SPEAKER> HAMLET </SPEAKER> <LINE> This? </LINE> </SPEECH>
    <SPEECH> <SPEAKER> First Clown </SPEAKER> <LINE> E'en that. </LINE>
      </SPEECH>
    <SPEECH> <SPEAKER> HAMLET </SPEAKER> <LINE> Let me see. </LINE>
      <STAGEDIR> Takes the skull </STAGEDIR>
      <LINE> Alas, poor Yorick! I knew him, Horatio: a fellow </LINE>
      <LINE> of infinite jest, of most excellent fancy: he hath </LINE>
      <LINE> borne me on his back a thousand times; and now, how </LINE>
      </SPEECH> </EXTRACT>  : SimpleXml.simple_xml
```

Before leaving prettyprinting altogether, let us look at the rest of the *Compiler.PrettyPrint* facilities. The structure defines an exception *PP_FAIL* which is raised for example when you terminate a block that you never opened. The call *add_newline* injects a forced linebreak into the *ppstream*, that is the prettyprinter will automatically break the line at that point. The call *clear_ppstream* clears the content of the *ppstream*, while *flush_ppstream* flushes the currently accumulated text in the stream, and moreover forces the flushing of the consumer. Given a *ppconsumer* (say, through *dest_ppstream*, to obtain the consumer of another stream), we can create a *ppstream* over that consumer through *with_pp* which creates a *ppstream* over the consumer and calls the supplied function with it. It is (essentially) equivalent to

```
fun with_pp c f = let
  val s = Compiler.Print.mk_ppstream (c)
in
  f (s);
  Compiler.Print.flush_ppstream (s)
end
```

Finally, the call *pp_to_string* prettyprints a value into a string. Going back to our *simple_xml* values example, writing a prettyprinter for the values displayed by the interactive loop is nice, but if one wants to output a *simple_xml* value into a string, one has to use the ugly *toString*, which among other things, loses all the structure of the underlying value. However, given a prettyprinter, we can call *pp_to_string* to prettyprint an expression into a string. In effect, we create a consumer to accumulate the output of the prettyprinter to the string. The function *pp_to_string* takes as arguments an integer giving the linewidth to use, a prettyprinter of type *ppstream* $\rightarrow$*'a* $\rightarrow$*unit* and a value of *'a*, and prettyprints that value into a string. As an example, compare

```
- print (SimpleXml.toString hamlet);
<EXTRACT><SPEECH><SPEAKER>First Clown </SPEAKER><LINE>A pestilence on him for a mad rogue! a' poured
```

with

```
- print (Compiler.PrettyPrint.pp_to_string 60 ppXml3 hamlet);
<EXTRACT>
  <SPEECH> <SPEAKER> First Clown </SPEAKER>
    <LINE> A pestilence on him for a mad rogue! a' poured a
      </LINE>
    <LINE> flagon of Rhenish on my head once. This same
      skull, </LINE>
    <LINE> sir, was Yorick's skull, the king's jester.
      </LINE> </SPEECH>
  <SPEECH> <SPEAKER> HAMLET </SPEAKER> <LINE> This? </LINE>
    </SPEECH>
  <SPEECH> <SPEAKER> First Clown </SPEAKER>
    <LINE> E'en that. </LINE> </SPEECH>
  <SPEECH> <SPEAKER> HAMLET </SPEAKER>
    <LINE> Let me see. </LINE>
    <STAGEDIR> Takes the skull </STAGEDIR>
    <LINE> Alas, poor Yorick! I knew him, Horatio: a fellow
      </LINE>
    <LINE> of infinite jest, of most excellent fancy: he
      hath </LINE>
    <LINE> borne me on his back a thousand times; and now,
      how </LINE> </SPEECH> </EXTRACT>val it = () : unit
```

5.4 Heap images

The interactive nature of the compiler is useful for writing short programs and testing functions, as well as providing a manner of command interpreter for programs can best be seen as a set of utility programs. However, standalone applications, a model like the one provided by compilers for most other languages, generating standalone object-code that does not require one to run the compiler again for execution is very useful. Among other things, it does not require a user of the application to have the SML/NJ compiler installed on his or her system.

Another problem with the interactive compiler (and it turns out to be a related problem) is that one may want to define a set of functions (utility functions or others) available at every invocation of the compiler, without necessarily having to recompile these utility functions through *use* or something of that ilk every time, like we did in Section 1.6.

Both the problems of creating standalone programs and customized versions of the compiler have the same solution, which we address in this section. First, a brief recap of heap images and how the SML/NJ memory model and compiler works.

The SML/NJ compiler is organized as a runtime system and a heap, with the heap containing all the data and the code used and generated by the compiler. In traditional compilers (batch compilers for languages such as C,C++,Pascal, etc), the code generated by the compiler is immediately sent to a file, called the object code file. A linker later turns all those object code files into a standalone executable program. In SML/NJ and other interactive compilers, the code generated is stored directly in memory, ready to be used by other code that will be compiled.

The runtime system is needed to provide an environment in which the generated code can execute. Object code in any language requires such a runtime system, sometimes in the form of a simple runtime library (such as the one for C, `libc`). SML/NJ's runtime supports garbage collection of the heap and the translation between SML and the underlying system's representation for system calls. The runtime system fundamentally is in charge of running the code in the heap. When the runtime system is started, it loads an initial heap and starts executing its code. By default, when SML/NJ starts up (by executing `sml`), it loads up a default heap image containing the compiler and starts executing the code in the heap. That code is simply a loop, querying the user for input, compiling the input, executing it, and looping for more input. To generate a standalone application (like the compiler), we need to create a heap that contains the code for the application and store it so that the code can be called by simply running the runtime system with a new heap. Similarly, creating a "customized" version of the compiler and environment amounts to saving a new heap containing the compieler and the code implementing the customization. We show how both of these work.

All of this is achieved through two functions in the structure *SMLofNJ*. The process of creating a heap image is called exporting a heap. The simplest function that exports a heap image is *SMLofNJ.exportML*. This function takes a string argument (the name of the file in which the heap will be written). and creates a heap image in the file containing a copy of the current heap, including all the existing identifier bindings up to the time of export. The result of *exportML* is interesting. After exporting the heap image, it returns the value *true* and the rest of the evaluation containing the call to *exportML* continues. Now, when the heap that was exported is subsequently loaded into the runtime system, the runtime system starts executing the code in the heap, at the point where *exportML* returns, even if *exportML* was embedded deep in some other expression. The difference is that this time *exportML* returns *false* and the rest of the code that was waiting after the result of *exportML* resumes evaluation (this code is of course stored in the heap and thus was saved when the heap was exported)[1]. Since part of the code containing the execution is the interactive loop (recall, it is the original piece of code that executes when the compiler is first invoked), we eventually return to the interactive loop and the compiler is available as before.

Let us consider a concrete example. Recall that in Section 1.6, we disucssed how various top-level definitions can help alleviate some of the difficulties remembering where various functions used frequently are located as well as greatly sim-

[1]This is a concrete example of a continuation, a subject we will discuss in much more detail in Chapter **??**.

plifying the navigation through the file system, providing a more “shell-like” experience. We discussed saving these definitions in a file `defs.sml` and to *use* this file every time the compiler is invoked. This is an ideal example for us: we will create a new heap image for the compiler, one which includes those definitions by default, removing the need to load `defs.sml` explicitely each and every time.

First, start a fresh instance of the compiler, and *use "def.sml"* to load and compile the definitions into the heap. Then, create a new version of the compiler by typing

```
- SMLofNJ.exportML "/home/riccardo/compdefs";
...
```

where of course you replace *"/home/riccardo/compdefs"* by the directory and filename you want to save the new heap image under (it can be anywhere). When *exportML* is done, you can exit the compiler.

Assuming everything went smoothly (you had access rights to the directory where you wanted to save the heap image, there was enough disk space, etc...), if you look into the directory you specified, there should be a rather large file called `compdefs.<something>` (or whatever the name was that you chose to save the file under). The `<something>` is a representation of the system you are using, and allows for heap images corresponding to different systems to coexist in a single installation. This is typically not an issue for the casual user. To use this heap image, you need to start the runtime system, specifying that this is the heap image you want to load instead of the default one. To achieve this, use the shell command:

```
sml @SMLload=/home/riccardo/compdefs
```

(where again `home/riccardo/compdefs` should be replaced by the appropriate path and filename. This will be assumed from now on.) This starts the runtime system, but the `@SMLload` flag tells the runtime system to load the specified heap image (note that there is no space before and after the =). At this point, the heap image has been loaded, the interactive compiler is running, and you have access to all the definitions that were in `defs.sml` without having to *use* the file. In fact, any definition you entered prior to the *exportML* will be there. Even better, any changes you had made to flags controlling the compiler or the runtime system will be restored.

Certain details are harder to get right. If you notice, the above heap, when run, start the compiler by simply displaying a value *false* (the result of *exportML*!) and prompting the user at the interactive loop. The default heap, when run, displays a banner identifying the compiler, before presenting the interactive loop. It is not that hard to implement such a feature, but it requires thinking carefully about what happens at the time of an *exportML*.

Consider what we want: we want to display a banner identifying the compiler when the exported heap is run. The first step is figuring out how to print the banner. Instead of creating it from scratch, we can simply use the banner printing by the compiler itself when it is loaded. The appropriate string is found in the structure *Compiler*. The string *Compiler.banner* contains the official SML/NJ compiler banner. All we need to do is print out this banner when the exported heap is loaded. How can we do that? Recall that when a heap exported by *exportML* is loaded, it continues executing the code that was waiting for the result of *exportML*, passing the value *false*. With this in mind, all we need to do is make sure that after *exportML*, we print the banner, in one fell swoop. Consider the following code:

```
- (SMLofNJ.exportML "/home/riccardo/compdefs2";
   print Compiler.banner;
   print "\n");
...
```

Executing this code exports a heap image `compdefs2`, which when loaded in the runtime system via `sml @SMLload=/home/riccardo/compdefs2` resumes the execution of the code at the point where *exportML* returns its value. In this case, since *exportML* is part of a sequence of operations, the next operation in the sequence executes, and prints the banner. Then the code finishes evaluating, returning *()* (the return value of *print*) and drops the user to the interactive loop, as desired.

A last small detail remains: we don't get the same behavior as the default compiler, since we have a spurious *val it = () : unit* before the interactive loop prompt. As we have seen, it is the return value of *print*, the last function in the sequence containing *exportML*. To do things right, we need to find a way to evaluate the sequence (solely for its side-effects) and not display anything as a result. This bit of SML/NJ trivia can answered with the following code:

```
- val () = ();
```

This piece of code performs a successful pattern-match on the result *()*, but since no variable gets bound, nothing gets displayed. Compare with the following:

```
- val (a,b) = (10,20);
val a = 10 : int
val b = 20 : int
```

Therefore, to get the "original compiler" effect, we can create the heap image as:

```
- val () = (SMLofNJ.exportML "/home/riccardo/compdefs3";
            print Compiler.banner;
            print "\n")
...
```

Being able to generate a new heap image to customize the environment is useful, but sometimes one may want to change the environment often, or dynamically decide what customization to perform. In those cases, the method outlined above is not the most effective. Moreover, since each exported image is around 5MB, having many different heap images around quickly fills space. We consider a different approach, although still implemented using the above method based on a model often found under Unix.

In Unix, applications will often define application-specific customizations in a special file that is examined every time the application is started. For example, the Unix shell *bash* reads a file `.bashrc` that contains customizations that can be changed by the user.[2]

Let us create a new compiler heap image which, when loaded, will check if a file called `.smlnjrc` exists in the user's home directory, and if so loads it in (it is assumed that the file will contains valid SML code) before starting the interactive loop. Clearly, one can put all the definitions in `defs.sml` in `.smlnjrc`, as well as some indication of what is going on, such as a line

```
print ".smlnjrc successfully loaded\n";
```

at the end of the file.

To implement this functionality, we define the function to actually load the file:

```
fun loadrc (s) =
```

One could also call the function *useSilently* defined in Section 5.2. Again, we are cheating, by putting a call to *use* inside the body of a function. In any case, we settle for the above and can now create a heap image:

```
- val () = (SMLofNJ.exportML "/home/riccardo/compdefs4";
            print Compiler.banner;
            print "\n";
            loadrc "/home/riccardo/.smlnjrc")
```

A more involved implementation would probably query environment variables to extract the user's home directory (the variable `HOME` for example).

There is a problem however, one that points to some unpleasantness in this process of creating customized versions of the environment. As we saw in the case of printing the banner, if we want to truly extend the compiler, we need to make sure that we reproduce exactly the functionality provided by the compiler at startup. Printing the banner is one thing performed at startup, but CM, the compilation manager which we study in detail in Chapter 6, adds wrinkles of its own. CM

[2]Under Windows, such customizations are often placed in the Registry. There is clear way at this point in time to access the Windows Registry from SML/NJ.

examines the command line used to invoke `sml`, checks if the name of a source file is given as an argument, and if so, loads it. It turns out that to reproduce the behavior of the compiler, we need code such as

```
val () = (SMLofNJ.exportML "/home/riccardo/compdefs6";
          print Compiler.banner;
          print "\n";
          CM.processCommandLine ();
          loadrc "/home/riccardo/.smlnjrc")
```

But of course, these features are highly version dependent and may change without notice. A good place to get a feel for the behavior of the system at startup is to look at the source code, in the scripts that build both the compiler and CM.

The ability to export a heap image which is a snapshot of the current heap has been successful in dealing with one of the problems, that of customizing the compiler with default declarations or behaviors (and indeed, most of the customization flags we encountered earlier in the chapter). Let us now turn to the other problem we mentioned at the beginning of this section, that of generating standalone code that does not require the user to go through the compiler to execute the application.

Suppose for the sake of argument that we have an application compiled in the heap, with an entry point function called *main*, of type *unit* →*unit*. That is, to execute the application, one invokes *main ()*, and the call returns when the application terminates. One way to generate an executable is to export a heap image as above, where the code executed at startup is simply a call to *main*:

```
- val () = (SMLofNJ.exportML "sampleapp1";
            main ())
...
```

This works nicely, in that if a user invokes `sml @SMLload=sampleapp1`, the application runs, and the application packager can create a little script that calls the above, and the only thing really needed is the runtime system and the heap image itself. We encounter two problems if we do things this way however. The first problem is that after the application executes, that is when the call to *main* returns, instead of nicely terminating the process, the user is dropped back into the interactive loop! We can remedy this by explicitely killing the SML/NJ process after the call to *main* (say, passing in a status returned by *main*, which we now assume has type *unit* →*OS.Process.status*):

```
- val () = (SMLofNJ.exportML "sampleapp2";
            OS.Process.exit (main ()))
...
```

But the second problem does not go away: although the exported heap image does not use the compiler (unless the application explicitely calls compiler-specific portions, which it probably does not), the exported heap image still contains a copy of the full SML/NJ compiler and environment, making the exported heap image huge, and wasting space and time. For example, using this approach, the typical HelloWorld program

```
fun main () = (print "Hello, world!\n";
               OS.Process.success)
```

generates an executable image of size close to 5MB, clearly excessive.

To solve this problem, we use a different heap export mechanism, the function *SMLofNJ.exportFn*. Just like *exportML*, *exportFn* exports a heap image, but rather than dumping a copy of the current complete heap image, it also takes as argument a function to be called when the heap is later loaded into the runtime system. When the function returns, the runtime system will terminate, thus never invoking the interactive loop, similarly to what we did above, although without us having anything special to do about it. But the more interesting thing that happens is that before exporting the heap, the system removes from the heap anything that does not have anything to do with the function mentioned in the *exportFn* call, only keeping in the heap the minimum required to execute the function. Thus, if the called function does not invoke the compiler, by never say referring to the *Compiler* structure, the compiler is removed from the heap and not exported. This results in drastically smaller heap images, much faster to load and execute. On the other hand, because the heap has been so reduced in size prior to the *exportFn* call, the interactive compiler cannot continue executing after an *exportFn*, and in fact *exportFn* terminates the process as well.

The function expected by *exportFn* has type *string* $\times$ *string list* $\rightarrow$ *OS.Process.status*. When the heap is loaded, the function gets called with the name of the command used to invoke the program (the one that ends up loading the heap) and a list of the command line arguments — the same arguments passed to the function *main* in C programs. The result value is used as the result passed back to the operating system when the process finishes. In effect, *exportFn ("foo",main)* is equivalent to

```
fun exportFn = (SMLofNJ.exportML "foo";
                OS.Process.exit (main (SMLofNJ.getCmdName (),
                                       SMLofNJ.getArgs ())));
```

except for that additional business of clearing the heap before performing the export. Note the functions *SMLofNJ.getCmdName* and *SMLofNJ.getArgs*, used to get at the command line arguments. We could have also used functions in the *CommandLine* structure of the Basis, for portability, although we are talking about exporting heap images, an intrinsically non-portable process. As a note, recall that *getArgs* does not pass in command lines arguments starting with `@SML`, which are directives to the runtime system. Thus, the function specified by *exportFn* also will not receive those arguments. A call to *SMLofNJ.getAllArgs* will however return the complete list of arguments provided to the process, including any runtime system directive.

In the rest of these notes, we will write most of our larger examples which are meaningful as standalone applications in a way suitable for use with *exportFn*. We will typically not mention this use of *exportFn* as the last step of the coding process, but we will always provide a structure *Main* matching the following signature:

```
sig
  val main : string * string list -> OS.Process.status
  val shell : string -> OS.Process.status
end
```

where *main* is the starting point function, passed to *exportFn*, and *shell* is a helper function for testing the application from SML/NJ's interactive loop, expecting a string representing an invocation of the application as it would be done from the shell. Using the string functions we saw in Section 4.2, it is a simple matter to define *shell*:

```
fun shell (s) = let
  val l = String.tokens Char.isSpace s
in
  main (hd (l), tl (l))
end
```

For most applications, part of the initial work performed in *main* is to parse and interpret the command line arguments. To help with such an endeavor, the SML/NJ Library provides a module *GetOpt* to process command line arguments. This library is described in more detail in Section 7.8.

5.5 Unsafe operations

As mentioned in the introduction, SML is a safe language, in the sense that the type system imposes a discipline that prevents errors that would cause programs to fail at run time. For example, a language such as C allows the conversion of an arbitrary integer into a pointer, and allows it to be dereferenced accordingly. No check is made to ensure that the pointer is valid or points to a valid part of memory. Typically, it does not, and thus a program using such a "feature" will die a horrible death at the hands of a segmentation fault.

On the other hand, the flexibility of being able to view values of a given type as values of another type can be useful, especially when dealing with very low-level data, or performing very low-level system programming, or simply when interfacing with low-level libraries written in other languages. The top-level structure *Unsafe*, whose signature is given in Figure 5.14, provides access to unsafe functionality. It goes without saying that such functionality should not be used lightly. Indeed, it is extremely easy to corrupt the heap with such functions, resulting in a crash of the system, nullifying the advantages of the SML type system.

```
structure Unsafe : sig
  structure CInterface : CINTERFACE
  structure Object : UNSAFE_OBJECT
  structure Vector : UNSAFE_VECTOR
  structure Array : UNSAFE_ARRAY
  structure CharVector : UNSAFE_MONO_VECTOR
  structure CharArray : UNSAFE_MONO_ARRAY
  structure Word8Vector : UNSAFE_MONO_VECTOR
  structure Word8Array : UNSAFE_MONO_ARRAY
  structure Real64Array : UNSAFE_MONO_ARRAY
  val getHdlr : unit -> exn cont
  val setHdlr : exn cont -> unit
  val getVar : unit -> 'a
  val setVar : 'a -> unit
  val getPseudo : int -> 'a
  val setPseudo : ('a * int) -> unit
  val boxed : 'a -> bool
  val cast : 'a -> 'b
  val pStruct : Object.object ref
  val topLevelCont : unit cont ref
end
```

Figure 5.14: The structure *Unsafe*

Let us examine the interesting functions in *Unsafe*. The functions *getHdlr* and *setHdlr* provide access to the current exception handler (an exception continuation, see Chapter **??**). Throwing an exception to that continuation will have the same effect as raising the exception. On a related note, the reference cell *topLevelCont* contains the current continuation for the top level. Throwing a unit to the continuation in the cell returns to the toplevel loop, aborting the current expression being evaluated. The function *cast* bypasses the type system, converting the type of a value without changing the underlying representation of the value. Various substructures are accessible through the *Unsafe* structure, handling unsafe versions of various kinds of vectors and arrays (unsafe in that there is no range checking on subscripts and so on), over which we will not go. The two most interesting substructures are *Object*, a general interface to object representations, and *CInterface*, the communication with the underlying native system.

The substructure *Unsafe.Object*, whose signature is given in Figure 5.15, provides functions to get at the underlying representation of values of various types. A complete understanding of the representation of values in compiled code requires at the very least knowledge of the compiler being used, but we can still make use of the functions. The structure defines an abstract type *object* for generic SML values, through which one can interrogate their representation. The function *toObject* creates such an interrogatable object from a given value of arbitrary type. The

```
structure Unsafe.Object : sig
  type object
  datatype representation
  = Unboxed
  | Real
  | Pair
  | Record
  | PolyArray
  | ByteVector
  | ByteArray
  | RealArray
  | Susp
  | WeakPtr
  val toObject : 'a -> object
  val boxed : object -> bool
  val unboxed : object -> bool
  val rep : object -> representation
  exception Representation
  val toTuple : object -> object vector
  val toString : object -> string
  val toRef : object -> object ref
  val toArray : object -> object array
  val toExn : object -> exn
  val toReal : object -> real
  val toInt : object -> int
  val toInt32 : object -> Int32.int
  val toWord : object -> Word.word
  val toWord8 : object -> Word8.word
  val toWord32 : object -> Word32.word
end
```

Figure 5.15: The structure *Unsafe.Object*

```
structure Unsafe.CInterface : sig
  exception CFunNotFound of string
  val c_function : string -> string -> 'a -> 'b
  type c_function
  val bindCFun : (string * string) -> c_function
  type system_const = (int * string)
  exception SysConstNotFound of string
  val findSysConst : (string * system_const list) -> system_const option
  val bindSysConst : (string * system_const list) -> system_const
end
```

Figure 5.16: The structure *Unsafe.CInterface*

functions *boxed* and *unboxed* check if the underlying representation of an object is respectively boxed (is is really a pointer) or unboxed (it is an immediate value). The function *rep* reports more information, returning a flag of type *representation*, indicating the representation of the value.

The remaining functions will convert the object to a specified type if compatible with the underlying representation. They raise the exception *Representation* otherwise. The function *toTuple* takes an object which is really a record, tuple, vector or real array and returns a vector of its fields; *toString* takes an object which is really a string and returns its string value; *toRef* takes an object which is really a reference cell or an array of length 1 and returns a reference cell; *toArray* converts an object which is really an array; *toExn* converts an object which is really an exception; *toReal* converts an object which is really a real number; *toInt* converts an object which is really a tagged 31-bit integer (type *Int.int*); *toInt32* takes an object which is really a 32-bit integer or a byte vector of length 4 into an *Int32.int* value; *toWord* takes an object which is really a tagged 31-bit value into a *word* value; *toWord8* takes an object which is really a tagged 31-bit value into a *Word8.word* value; and *toWord32* takes an object which is really a 32-bit value or a byte vector of length 4 into a *Word32.word* value.

Most of the uses of the functions in *Object*, if not for hacking compiler tools, is to deal with interfacing with the underlying runtime system, through the foreign function interface. The foreign function interface for SML/NJ is not as easy to use as some other environments, but upcoming versions of the system should greatly improve the situation. However, most improvements will take the form of additional, easier to use layers over the foundations provided by the interface in substructure *Unsafe.CInterface*, whose signature is given in Figure 5.16.

Notes

The tradition of interactive compilation seems to originate with Lisp, one of the original mostly-functional languages. (Another early language that has benefitted from interactive interfaces was SmallTalk.) SML is in fact in the tradition of Lisp, except for the fact that SML introduces static typing instead of the dynamic typing of Lisp. The modern equivalent of Lisp is Scheme [56], which also inherits the traditional S-expression syntax. Compilers for Scheme are also typically interactive, for instance the PLT compiler project from Rice University, the MIT Scheme compiler from MIT, Scheme48 from NEC Research Labs. Scheme has also been adopted by GNU as a macro language, under the name Guile [**?**].

Runtime systems for languages and compilers are often the least documented aspect of a system. In the case of SML/NJ, descriptions of early versions of the runtime system can be found in Appel's book [5] and paper [4]. The details are rather dated, but should give an idea of the overall structure of the runtime system. Garbage collection is the most complex part of the runtime, and is usually better described. The current version of SML/NJ uses a generational garbage collector, described mostly in [93]. A good overview of (simple-processor) garbage collection algorithms can be found in [114].

The prettyprinting mechanism described and used by SML/NJ is due to Oppen [85]. It is simple to implement, but not very flexible. It is sufficient for most common uses of prettyprinting, but for more complex jobs, a prettyprinter such as PPML [77] should be used. A library for prettyprinting based on PPML primitives is being provided in the SML/NJ Library, and should be documented in a future version of these notes. Prettyprinting algorithms based on a library of functional combinators can be found in [51, 113].

The unsafe interface is typically useful when interfacing with very low-level code, such as when performing calls to the underlying runtime system, or to external C functions. The current foreign function interface (FFI) of SML/NJ is not very well documented, and awkward to use (it requires a recompilation of the runtime system). For Unix systems, an easy-to-use user library to ease the interfacing of foreign-code, the C-calls library [50], is available. It still requires a recompilation of the runtime system to add new code, but it greatly simplifies the communication of values back and forth from SML to the foreign code. An improved version of the FFI, based on an Interface Definition Language (IDL) [118], is in the works [90]. This should bring the system in line with other functional languages implementations providing such an interface, such [30] and [62].

Chapter 6

The Compilation Manager

Until now, we have been feeding our programs into the compiler in either of two ways: directly at the interactive loop, or by loading files via *use*. The drawback of entering programs directly at the interactive loop is clear. The interactive loop is useful for testing functions and trying out algorithms and idea, but as a software engineering tool, it leaves much to be desired. Loading files through *use* is already much better, but for large projects it is extremely inefficient: every time a project is loaded via *use*, all the files in the project are read and compiled. Moreover, if a change to source file is made, all of the files making up the project need to be reloaded and recompiled. We do not get incremental compilation.

In the Unix world, selective and incremental recompilation is achieved through the use of a tool such as *make*, which takes a simple file describing the dependencies amongst the files in the project, as well as the tools to invoke, and rebuilds the appropriate parts of the project that need rebuilding. The idea is to check which files have changed by looking at the time stamp of the files, and a file is rebuilt if the time stamp of the target (if it exists) is older than any of the files it depends on. Note that the user needs to specify by hand the dependencies between the files (although in some instances a tolls such as *makedepend* can be used), and the user has to specify the precedure to rebuild out-of-date files (although appropriate defaults often exist).

A tool such as *make* is language-independent. It can be used to drive many different compilers, and indeed it could even be used to drive SML/NJ. One problem is that *make* is strongly batch-oriented, while SML/NJ is an interactive compiler. SML/NJ could be used as a batch compiler, but since there is no real support for compiling code to a file instead of to memory, it would be difficult to do cleanly, given the current infrastructure.

The approach that has been taken by the SML/NJ developers is to incorporate

```
signature HELLO_WORLD = sig
  val main : unit -> unit
end

structure HelloWorld : HELLO_WORLD = struct
  fun helloworld () = print "Hello World!\n"
end
```

Figure 6.1: The Hello World program

the functionality of a *make*-like tool directly in the compiler. One advantage of pushing the building tool inside the compiler and making it SML-specific is that we can perform much more precise dependency checking and most importantly automatic dependency checking when the times come to determine which files to recompile in a project.

The tool we describe in this chapter, the SML/NJ Compilation Manager (CM for short), should be used for any project larger than a file or two. Handling of libraries is also done through CM, and indeed most of the code we will see from now on in these notes will assume that one knows how to use CM.

6.1 Overview of CM

The simplest way to use CM is through a description file (which we call the root file) that lists the source files that make up a project. CM automatically tracks dependencies between the files listed in the root file. Because dependencies are only tracked across structures, the SML source files listed in the root file should contain only structure, signature or functor declarations. This is not a big restriction, since we already established in Chapter 3 that all the entities in an application should be at the level of the module system.

To keep the discussion concrete, consider a simple example. Figure 6.1 gives a structure and a signature for a trivial application, an overgrown Hello World program. We assume that structure *HelloWorld* has been put in file `helloworld.sml` and signature *HELLO_WORLD* has been put in file `helloworld-sig.sml`. To compile the program, instead of doing something along the lines of:

```
- (use "helloworld-sig.sml"; use "helloworld.sml");
...
```

we instead create a file `sources.cm` in the same directory as `helloworld.sml` and `helloworld-sig.sml`, containing the following:

```
Group is
```

```
helloworld-sig.sml
helloworld.sml
```

This file is a CM description file, stating that the application consists of the given files. (The *Group* annotation is to indicate that the description file defines an application, as opposed to a library. This has consequences with respect to how compiled symbols are exported. More on this later.)

To compile a project with such a description file, make sure the compiler's working directory is set to the directory containing the file `sources.cm` (possibly using *OS.FileSys.chDir* to change directories), and type:

```
- CM.make ();
...
```

The function *CM.make* reads the file `sources.cm` and in turns compiles the required files into memory. After the code is compiled, the initialization code of the structures is executed. The initialization code for a structure includes all the value bindings in the structure, which must be evaluated when the structure is declared. If the compiler determines that one of the files does not need to be recompiled, because it has not changed and none of the other files on which it depends have changed in any way that affects it, it will not be recompiled. Thus if one immediately does a:

```
- CM.make ();
...
```

we see that the code has not been recompiled.

The function *CM.make* compiles a CM description file which is named `sources.cm` in the current working directory. To compile a CM description file in another directory or with a different name, one can use the more general *CM.make'* function, which takes a string argument specifying a path and a CM description file. For example, one could invoke *CM.make' "path/to/foo.cm"*. In general, functions in the *CM* structure ending with ' require an explicit filename argument, while the corresponding functions without the ' work on the default description file `sources.cm`. It is possible to change the name of the default description file by calling *CM.set_root* with the new default.

The functionality of CM is accessed through the *CM* structure, whose partial signature is given in Figure 6.2 (the actual signature is more extensive, but we will only cover the basic functionality here). Let's give a quick overview of some of the rest of the functionality. Calls to *recompile* (and *recompile'*) perform a recompilation of the code just like *make*, but do not execute the initialization code of the structures, nor do they introduce new bindings in the environment. Moreover, the compiled code is not kept in memory, but rather compiled into files, which are used as a cache.

```
structure CM : sig

    structure Tools: TOOLS

    val version: string
    val verbose: bool option -> bool
    val debug: bool option -> bool
    val keep_going: bool option -> bool
    val parse_caching: int option -> int
    val show_exports: bool option -> bool
    val set_root: string -> unit
    val set_path: string list option -> string list

    val make': string -> unit
    val make: unit -> unit

    val recompile': string -> unit
    val recompile: unit -> unit

    val mkusefile': string * string -> unit
    val mkusefile: string -> unit

    val sa': string * string -> unit
    val sa: string -> unit

    val stabilize': string * bool -> unit
    val stabilize: bool -> unit
    val destabilize': string -> unit
    val destabilize: unit -> unit

    val autoload': string -> unit
    val autoload: unit -> unit
    val autoloading: bool option -> bool
    val clearAutoList: unit -> unit
    val autoList: unit -> string list

    val sweep: unit -> unit
    val clear: unit -> unit

    val procCmdLine: unit -> unit
end
```

Figure 6.2: The structure *CM*

As we mentionned above, *set_root* changes the default root file name used by *make* and *recompile*. The operating system environment variable `CM_ROOT` also controls this default, when CM is initially loaded (when SML/NJ starts).

When it is possible to do so, the function *mkusefile* (and *mkusefile'*) creates a SML file containing a sequence of *use* commands to load the source files specified in the root file. The idea is simply to topologically sort the files so that any file f_1 that depends on file f_2 is loaded after f_2 in the sequence of *use* commands. All the examples we have seen until now can be easily linearized in that way, and indeed that will be the case in general. This mechanism breaks down when we consider export filters and subgroups in Section 6.2, with which one can write programs that are not linearizable.

As an alternative, one can use the function *sa* (and *sa'*) to produce a SML program that loads all the source files just as in the *mkusefile* case, except that the program manages the namespace the same way CM does (which we will see in Section 6.2), and moreover attempts to load the binfiles (the object files) when they exist, avoiding recompilation. The function *sa* produces a program which relies on CM to perform it loading action. On the other hand, *mkUseFile* produces a file which can be used with other SML compilers, provided they support the *use* function.

6.2 Group hierarchies

The examples of the previous section showcase the usefulness of CM to manage the compilation and recompilation of applications, but fails to even hint at the available power to express modular structure. By default, every structure defined in a group is available after a *CM.make ()*. In this section, we introduce the notion of export filters to specify exactly which structures (signatures, functors) are available after compilation. Consider the following code:

```
signature A_SIG = sig
  val main : unit -> unit
end

structure A : A_SIG = struct
  fun main () = B.bar ()
end

structure B = struct
  fun bar () = print "Hello\n"
end
```

split across files “a-sig.sml”, “a.sml” and “b.sml”. The main entry point is structure *A*, while *B* handles support. To hide *B* after compilation, we specify an explicit export filter in “sources.cm”:

```
Group
  structure A
  signature A_SIG
in
  a-sig.sml
  a.sml
  b.sml
end
```

and indeed, we can try:

```
- CM.make ();
val it = () : unit
- A.main ();
Hello
val it = () : unit
- B.bar ();
stdIn:21.1-21.6 Error: unbound structure: B in path B.bar
```

Restricting access to elements is a powerful aspect of modularity, as we saw in the case of the module system. In some way, we can view the export filter as a kind of super-signature in a super-module system, with different memory and linking mechanisms.

Another aspect of modularity, aside from data hiding, is packaging. For example, it is clear that if we write a set of modules meant to be accessed by many applications (for example, the stacks and queues of Chapter 3), one would not want to always specify the names of the files to be loaded in every CM description file that happens to need to use stacks and queues. CM supports the notion of subgroup. That is, it is possible to specify in a CM description file another description file that will be loaded along with the rest of the source files, compiling its corresponding files and so on.This process is called importing a group.

One approach to dealing with components is to create a CM description file for each component, and list in the description file of the main application both the files of the application itself and the various description files for the components. It is not even necessary to know the direct path to those description files, as they will be looked up through the `CM_PATH` environment variable. Any library of use will have its location added to the path (later, we will see an alternative way of managing libraries, through the use of aliases).

The principal subtlety to be aware of when importing groups is how the exported symbols get managed. Recall that by default, unless an export filter is specified, every binding in a group gets exported. The generalization of this rule to imported groups is clear. By default, every binding in an imported group gets exported, unless an export filter is specified. The exports of an imported group are handled as if they were part of the importing group, and treated as such for the purpose of determining the exports of the importing group. Moreover, a group can define bindings of the same name as those exported by an imported group, effectively masking the underlying definition (of course, the original definition is still available within the imported subgroup).

The use of imported groups along with export filters can lead to code which is not compilable without CM, in the sense that no linearization via a sequence of *use* calls can create an equivalent result. For example, consider two files `f1.sml` and `f2.sml` which both define structures *A* and *B*. Suppose we have a file `g.sml` which wants to use structure *A* from `f1.sml` and structure *B* from `f2.sml`. Clearly, no linear ordering of *use* can have that effect. Using CM, we can create CM description files `f1.cm`:

```
Group
  structure A
is
  f1.sml
```

and `f2.cm`:

```
Group
  structure B
is
  f2.sml
```

and finally define a CM description file `g.cm`:

```
Group is
  g.sml
  f1.cm
  f2.cm
```

to get the desired effect of loading the file `g.sml` with the right combination of *A* and *B*. Of course, in general, instead of having files `f1.sml` and `f2.sml` providing similar structures, we have groups `f1.cm` and `f2.cm` exporting similar structures. If we want again the file `g.sml` to use structure *A* from group `f1.cm` and structure *B* from group `f2.cm`, we can simply write proxy CM description files whose only role is to wrap export filters around the original group files, such as files `f1p.cm`:

```
Group
  structure A
is
  f1.cm
```

and `f2p.cm`:

```
Group
  structure B
is
  f2.cm
```

along with a main `g.cm` CM description file:

```
Group is
  g.sml
  f1p.cm
  f2p.cm
```

There is a slightly annoying problem with groups as defined above. When a group loads a subgroup, every binding defined in the subgroup gets exported by the group, unless an export filter is defined. If the subgroup is used exclusively to help implement functionality in the group, this can be annoying. This forces one to write an export filter for the group. Since a heavy use of libraries leads to a lot of situations like that, we end up writing export filters for every group, or not caring that the symbols are exported. To balance this situation, a new kind of group, called a library, is defined. A library description file is a CM description file of the form:

```
Library
  structure A
  signature B
  ...
is
  file1.sml
  file2.sml
  ...
```

A library is required to provide an export filter, unlike groups. A library behaves just like a group, except when it is imported by a group. The definitions of the library are accessible from the sources of the importing group, but are not exported by the importing group by default. They can be exported if the importing group specifies them in an explicit export filter.

6.3 Tools

Not every file in a project is an SML source file. Some files contain descriptions that are translated into SML code using various tools which process the descriptions. The standard examples of such tools are ML-Lex and ML-Yacc (described in Chapters **??** and **??**), which take a declarative description of lexical tokens and grammar rules respectively and translate them into SML code for lexers and parsers. CM can be used to automatically apply the appropriate tools to description files.

CM uses the concept of a tool class to determine how to process a file. A tool class specifies a particular processor to use, along with a rule for determining the target file names from the source file names, as well as a set of conditions under which the processor is invoked. Typically, a condition is of the form “if one of the targets is out of date or missing”, that is if one of the target files has a modification date which is earlier than the source file (indicating that the source file has been modified since the last tool invocation), or if one of the target files is actually missing.

Builtin tool classes that CM knows include:

SML	the class of SML source files (no processor required)
CM	the class of CM description files (no processor required)
MLYacc	the class of ML-Yacc grammar files (requires ML-Yacc)
MLLex	the class of ML-Lex lexer files (requires ML-Lex)
RCS	The class of RCS files (requires checkout tool *co*)

Determining to which tool class a given file belongs to is typically done by looking at the suffix of the files. Default suffixes include:

.sig, .sml, .fun	SML class
.cm	CM class
.grm,.y	MLYacc class
.lex,.l	MLLex class
,v	RCS class

It is also possible to explicitely state which tool class a file belongs to in a CM description file by adding the tool class after the file name, as in:

```
not-a-grammar.grm : Sml
```

which will view the file `not-a-grammar.grm` as belonging to class *Sml*, and thus not requiring any processing and loadable directly.

In each case where a tool is applied, the files that result from the application of the tool are considered again for tool application. For example, if `f.grm` is present in a CM description file, ML-Yacc is invoked and generates files `f.grm.sig` and `f.grm.sml`, which are SML source files and thus processed directly by CM. In contrast, if `f.grm,v` appears in a CM description file, RCS checkout is invoked to create a new version of `f.grm`, which CM recognized as a ML-Yacc source file requiring the invocation of ML-Yacc to produce again `f.grm.sig` and `f.grm.sml`.

It is fairly straightforward to add new tool classes to CM, without recompiling CM. On the other hand, the modification needs to be done in CM, since the description file can only specify a tool class, and CM needs to know about the tool class to be able to invoke the appropriate tool (as opposed to *make*, which does not require modifying *make* to use new tools). The substructure *CM.Tools* (signature given in Figure 6.3) provides the required functionality.

As we said earlier, tools are associated with classes that describe how to process various kind of files. A class is described by four components:

1. a name, a string of lowercase letters;

2. a rule, from source name to target names. A source name is typically a filename, but really can be anything. A target is a name of something produced by the tool along with an optional class name stating the class of the target;

```
structure CM.Tools =   sig
  type abstarget = ?.AbsPath.t * class option
  type class = string
  datatype classification
    = CMFILE
    | SCGROUP
    | SCLIBRARY
    | SMLSOURCE
    | TOOLINPUT of {make:unit -> unit, targets:abstarget list,
                    validate:unit -> bool}
  datatype classifier
    = GEN_CLASSIFIER of fname -> class option
    | SFX_CLASSIFIER of string -> class option
  type fname = string
  type processor = {source:fname, targets:target list} -> unit
  type rule = fname * rulecontext -> target list
  type rulecontext = rulefn -> target list
  type rulefn = unit -> target list
  type simplerule = fname -> target list
  type target = fname * class option
  type validator = {source:fname, targets:target list} -> bool
  exception ToolError of {msg:string, tool:string}
  exception UnknownClass of class
  val addClassifier : classifier -> unit
  val addCmClass : class -> unit
  val addScGClass : class -> unit
  val addScLClass : class -> unit
  val addSmlClass : class -> unit
  val addToolClass : {class:class, processor:processor, rule:rule,
                      validator:validator}
                     -> unit
  val classify : ?.AbsPath.t * class option -> classification
  val defaultClassOf : fname -> class option
  val dontcare : simplerule -> rule
  val stdExistenceValidator : validator
  val stdSfxClassifier : {class:class, sfx:string} -> classifier
  val stdShellProcessor : {command:string, tool:string} -> processor
  val stdTStampValidator : validator
  val withcontext : simplerule -> rule
end
```

Figure 6.3: The structure *CM.Tools*

3. a validator, taking a source name and the target list produced by the rule, and determining whether the tool need to be invoked.

4. a processor, to actually implement the tool, that is mapping the source name to targets.

Although member names (both source and target) need not be filenames, they very nearly always are, and thus to keep the presentation simple I will refer to them as filenames.

Adding a new tool class to CM is simply a matter of calling *CM.Tools.addToolClass*, passing in the class name, the rule, the validator and the processor. Let us get the types right first:

```
type fname = string
type class = string
type target = fname * class option
```

that is filenames and classes are simply strings, and a target is a string with an optional associated class. Validators and processors are functions taking as argument the source filename and a list of targets. Note that validators and processors are passed the names as they appear in the CM description file.

```
type validator : {source : fname, targets: target list} -> bool
type processor : {source : fname, targets: target list} -> unit
```

Since the relative paths are resolved relative to the directory the description file appears in, CM temporarily changes its working directory to the directory the description file appears in , to simplify processing. This temporary working directory is called the “context”.

The last element required for a tool class is a rule. Rules, as we saw, take a source filename and generate a list of targets. Their interface is more complicated than validators and processors, due to additional flexibility: since rules often do not require the context to be set to work correctly (the name of the targets is often derived from the name of the source, without actually needing to read the source file), and since setting up the context can be expensive (for example, on network drives), the interface to rules allows the programmer to specify whether or not a context should be set up. The type of a rule is clear:

```
type rule = fname * rulecontext -> target list
```

it takes a source filename and a rule context, and returns a target list. The rule context is used to represent the idea that a context can be setup if desired. It has type:

```
type rulefn = unit -> target list
type rulecontext = rulefn -> target list
```

If a rule does not care about the context, the second argument to the rule can be ignored. Otherwise, the rule can pass a *rulefn* function to the rule context to use the *rulefn* in the appropriate context. For example, a simple rule which says that a source `x.src` produces a target `x.src.sml` does not require a context (since the target can be determined without access the filesystem), and is easily implemented as:

```
fun addSmlRule (fname,rulecontext) = [(fname^".sml",NONE)]
```

Notice that we did not specify a tool class for the target, relying on the default behavior for files with a `.sml` suffix. We could also have specified *SOME "Sml"* as a tool class for the target, with the same effect.

Suppose now that we wanted to implement a rule taking a source file `x.src` and producing a target `y.sml` where the name `y` is taken from the first line of the file `x.sml`. Since this requires accessing the file, and the filename (if relative) is as always given relative to the directory the CM description file appears in , we need to have CM set up a context for us. The rule therefore must call its *rulecontext* argument. Here is one way of achieving this (note that there is no error processing in this example):

```
fun readNameRule (fname,rulecontext) = let
  fun doit () = let
    val in = TextIO.openIn (fname)
    val name = TextIO.readLine (in)
    val name = String.extract (name,0,SOME (size (name) - 1))
  in
    TextIO.closeIn (in);
    [(name^".sml",NONE)]
  end
in
  rulecontext (doit)
end
```

Some of the messiness in the code has to do with reading the actual name from the file, namely to remove the trailing newline at the end of the string read in by *TextIO.readLine*. Some amount of lexing is typically necessary to do this cleanly (a simple token reader reading all characters up to the first whitespace, for example):

```
fun tokenReader (string) = ...
```

(For the interested reader, more advanced ways of doing such things will be described in Chapter **??** when discussing regular expressions.)

It is typically the case, as witnessed by the above, that a rule either always uses its context, or never does. Provisions are made in *CM.Tools* to handle such cases, called simple rules. A simple rule has type:

```
type simplerule = fname -> target list
```

and a simple rule can be converted to a rule by one of:

```
val dontcare : simplerule -> rule
val withcontext : simplerule -> rule
```

A rule created by *dontcare* never needs its context, and indeed *dontcare f* is equivalent to *fn (fname,_) => f (fname)*. On the other hand, a rule created by *withcontext* always sets up its context, and *withcontext f* is equivalent to *fn (fname,c) => c (f fname)*.

The *CM.Tools* substructure provides functions to create common validators and processors. Two standard validators are:

```
val stdTStampValidator: validator
val stdExistenceValidator: validator
```

that respectively verify time stamp consistency (the source is more recent than the targets) and the existence of targets. The most common type of processor is a shell command passed the source name as an argument. Such a processor can be created by calling

```
val stdShellProcessor : {command: string, tool:string} -> processor
```

where *command* is the shell command to execute, and *tool* is the name of the tol used for error reporting. Errors arising from tools should be reported by raising the exception *CM.Tools.ToolError*, which takes a value of type {*msg: string, tool: string*} where *tool* is the representation of the tool name and *msg* contains an explanation of the error.

We have seen how to define new tool classes and add them to the Compilation Manager. At this point, the only way to use such a tool class is to explicitely specify in a CM description file that such and such name are to be processed by that class, using the *: <class>* annotation. To automatically invoke the tool class for various forms of names, we need to define a classifier for the tool class. In effect, classifiers are used to try to determine the tool class associated with a name from the form of the name alone, typically using its suffix or its extension.

Two types of classifiers are defined. The simplest classifier (*SFX_CLASSIFIER*) is a function taking a suffix of the name to be classified and returning *SOME (c)*, with *c* a class name, if the suffix classifies the file as being processed by tool class *c*, or *NONE* if the suffix is not recognized by the classifier. A more general classifier (*GEN_CLASSIFIER*) takes the whole file as input and again returns a *class option* result. When CM finds a file name in a CM description file (or as a target from applying a rule), it attempts to discover the tool class of the file by calling every classifier registered with the system in turn, in some unspecified order, until on the them returns *SOME (c)*, at which point it invokes the rule, validator and processor defined by *c* on the name.If all the classifiers return *NONE*, an error is reported.

A new classifier is defined by calling either *SFX_CLASSIFIER* with a *string →class option* as an argument, or *GEN_CLASSIFIER* with a *fname →class option* as an argument. For the purist, note that *SFX_CLASSIFIER* and *GEN_CLASSIFIER* are in fact data constructors for the *classifier* datatype. A classifier so defined can be registered with the system by calling *addClassifier*, passing in the classifier. The function *defaultClassOf* will invoke the classification mechanism described above on a given file; this can be useful if the classification depends on the classification of other parts of the name. Note that classification does not change the context and it is usually a bad idea to go to the filesystem during classification — although classifying based on the content of a file is a powerful idea.

Since most classifiers simply look for a standard filename suffix, *CM.Tools* provides a convenient function to create such a classifier:

```
val stdSfxClassifier : {sfx: string, class: class} -> classifier
```

where *sfx* is the suffix to look for and *class* is the corresponding tool class. If a tool can handle files with two kind of suffixes or more, a standard suffix classifier for each recognized suffix can be registered with the system. For example, as we saw, the tool class *MLYacc* can process files with extension `.y` and `.grm`. This could be expressed as follows:

```
addClassifier (StdSfxClassifier {sfx="y",class="mlyacc"})
addClassifier (StdSfxClassifier {sfx="grm",class="mlyacc"})
```

6.4 A simple configuration tool

Let us add a very simple tool to CM. To simplify the example, we will not implement the processor as a separate application (such as ML-Lex or ML-Yacc), but rather as an internal function. This is just for illustration purposes, as most real uses of the tools facility should refer to external tools — if only for the reason that we should still be able to apply the tools externally, without running CM. In any case, we describe a simple tool to configure some files according to configuration variables, in a way reminiscent of *AutoConf* files under Unix. Here is the idea. We are given a source file which contains instances of configuration variables of the form @@*xyz*@@ for *xyz* a sequence of alphanumeric characters. When such a file is encountered in a CM description file, the system should replace all occurences of @@*xyz*@@ by a user-specified string, producing a complete source file that can then be compiled accordingly. Configuration variables are managed from the toplevel loop of SML/NJ. The user is provided with a structure *Cfg* to handle the setting of configuration variables.

```
signature LOOKUP_TABLE = sig
  type table
  val new : unit -> table
  val get : table * string -> string option
  val set : table * string * string -> unit
  val foldl : ((string * string) * 'c -> 'c) -> 'c -> table -> 'c
end
```

Figure 6.4: The signature *LOOKUP_TABLE*

```
structure NaiveLookupTable : LOOKUP_TABLE = struct
  type table = (string * string ref) list ref

  fun new () = ref []

  fun get (t,s) =
    (case List.find (fn (s',_) => s=s') (!t)
       of NONE => NONE
        | SOME (_,sr) => SOME (!sr))

  fun set (t,s,v) =
    (case List.find (fn (s',_) => s=s') (!t)
       of NONE => t := (s,ref (v))::(!t)
        | SOME (_,sr) => sr := v)

  fun foldl f b t = List.foldl (fn ((s,sr),r) => f ((s,!sr),r)) b (!t)

end
```

Figure 6.5: The structure *NaiveLookupTable*

For reasons made clear later, we create a CM description file `mlconfig.cm` that contains the actual implementation of the tool. Here it is in its exceeding simplicity:

```
Group is
  cfg.sml
  mlconfig.sml
```

The code is logically split into two parts, reflected in the two files: the first (`cfg.sml`) is the implementation of the configuration variables lookup table, the second (`mlconfig.sml`) is the implementation of the tool proper, which walks over a given source file performing the replacement. We consider these parts in turn.

Our implementation of the lookup table is naive. As we shall see in Chapter 7, SML/NJ has libraries to make this much more efficient and with an eye towards that, we functorize out tool over the implementation of lookup tables. Our

```
functor CfgFun (structure T : LOOKUP_TABLE) = struct
  val cfgVars = T.new ()
  fun set (s,s') = T.set (cfgVars,s,s')
  fun get (s) = T.get (cfgVars,s)
  fun list () = T.foldl (fn ((s,s'),_) => (print (s);
                                           print " = ";
                                           print (s');
                                           print "\n")) () cfgVars
end
```

Figure 6.6: The functor *CfgFun*

structure *NaiveLookupTable* (in Figure 6.5) is a mapping from strings to strings. Lookup tables are implemented as imperative maps, where updating a lookup table is destructive operation on the lookup table. A table is simply a list of association between strings and references to strings. The fact that the value associated with a string is a string reference means that the associated value can be updated in place. The functor *CfgFun* (Figure 6.6) creates a structure that can be used to access the configuration variables. To actually construct a structure *Cfg*, we instantiate the functor:

```
structure Cfg = CfgFun (structure T = NaiveLookupTable)
```

The tool proper is implemented in structure *MLConfig*, which provides a single entry point *processFile*. It uses the *Cfg* structure to lookup its variables. Walking the file is done by reading in each line successively, and looking for configuration variables. An alternative way would be to suck in the whole file at once (through *TextIO.inputAll*) but that requires making sure the file is not too large. Alternatively, we could suck in whole bufferfulls of the file. I leave this approach as an exercise for the reader. I focus on the straightforward if less efficient implementation. The function *processFile* simply opens the given file, creates the output file and for each line of the source file calls *processLine*:

```
fun processFile (file_in,file_out) = let
  val in = TextIO.inputIn (file_in)
  val out = TextIO.inputOut (file_out)
  fun loop () = (case TextIO.readLine (in)
                   of "" => ()
                    | s => (TextIO.output (out,processLine (s));
                           loop ()))
in
  loop ();
  TextIO.closeIn (in);
  TextIO.closeOut (out)
end
```

Processing a line is done using substring magic. It first finds the leftmost @@, and the leftmost @@ following it, and replaces the content by the string associated to the corresponding configuration variable:

```
fun processLine (s) = let
    val ss = Substring.all (s)
    fun findAtAt (ss) = let
      val (pref,suff) = Substring.position "@@" ss
    in
      if (Substring.isEmpty (suff))
        then NONE
      else SOME (pref,Substring.triml 2 suff)
    end
    fun replVarLoop (ss) =
         (case findAtAt (ss)
            of NONE => [ss]
             | SOME (s1,s3) => (case findAtAt (s3)
                                  of NONE => [ss]
                                   | SOME (s1',s3') =>
                                    (case Cfg.get (Substring.string (s1'))
                                       of NONE => raise ConfigError
                                        | SOME (s) => s1::Substring.all (s)::replVarLoop (s3'))))
  in
    Substring.concat (replVarLoop (ss))
  end
```

At this point, it is possible to test what has been done. Create a sample file `test.sml.cfg` with content:

```
structure Test = struct
  fun main () = (print "This file was created by @@name@@\n";
                 print "on @@date@@\n")
end
```

Compile the configuration tool we wrote above by typing *CM.make' "mlconfig.cm"*. Set the appropriate configuration variables, such as:

```
- Cfg.set ("name","Riccardo Pucella");
val it = () : unit
- Cfg.set ("date","August 17, 2000");
val it = () : unit
```

Finally, invoke the configuration tool on the sample file:

```
-
- MLConfig.processFile ("test.sml.cfg","test.sml");
val it = () : unit
```

This should produce a file called `test.sml` in the working directory, with the configuration variables instantiated to whatever you chose above.

Let us now write the code to install MLConfig as a tool in CM. For technical reasons, it is not easy to install the tool automatically via CM say by compiling a module which performs as a side effect the registration of the tool class[1]. Thus,

[1]The problem is that by default CM is not aware of itself! So accessing the *CM.Tools* structure from a file compiled by CM fails.

although we still describe the process as a module, the module should be compiled directly at the interactive loop (say via *use*, one the few such uses we will ever consider). This is not a very big deal, since anyone serious about this would then export a new heap image containing the modified CM (see Section 5.4).

With this in mind, we consider a file `install-mlconfig.sml` consisting of a simple structure *InstallMLConfig*, which should be compiled in an environment containing *MLConfig* and *Cfg*. The structure is in charge of registering the tool class corresponding to *MLConfig*. The code for the structure is presented in Figure 6.7.

The first step consists of creating a tool class, that is selecting an appropriate rule, validator and processor. The rule itself is simple: it takes the source file name and removes the `.cfg` extension, if it is present, and raises an exception otherwise. The function *stripCfgExt* takes a filename and removes the `.cfg` extension (it raises an exception *ConfigError* if no extension can be found). It uses substrings to perform the appropriate lookups. The resulting target is assigned a default tool class. Note that the rule does not need a context (since it does not access the file system), so we can create the rule through a call to *CM.Tools.dontcare*. The validator used is simply the time stamp validator, since the file should be run though the configuration tool whenever the source is modified. The processor is simply a call to *MLConfig.processFile*, with a suitable arrangement of arguments. Finally, we can wrap it all up, including a classifier that automatically classifies files according to the `.cfg` extension.

To add the tool to CM, it suffices to compile the above module, via say

```
- use "install-mlconfig.sml";
...
```

And this registers the tool. Create a sample CM description file for the above ”test.sml.cfg” example, something simple such as the following, saved under the name ”test.cm” (the name ”sources.cm” may already be taken by the tool in the directory):

```
Group is
  test.sml.cfg
```

and compile it using *CM.make' ”test.cm”*. The MLConfig tool should be invoked properly, and the resulting file ”test.sml” should then be loaded. As an exercise for the reader, it may be interesting to add some verbose output to the tool to recognize that it is indeed being run.

The main problem with this approach is clear: the tool must be registered with CM in order for the description file to make sense to the system. Of course, it is a simple matter to load the tool, but a user wanting to compile the file needs to go through the trouble of loading the tool explicitly, via the prompt. A much

```
structure InstallMLConfig = struct

  val toolName = "mlconfig"
  val class = "mlconfig"

  fun stripCfgExt (f) = let
    val cfg = Substring.all ("cfg")
    val ss = Substring.all (f)
    val (l,r) = Substring.splitr (fn c => not (c = #".")) ss
  in
    case Substring.compare (r,cfg)
      of EQUAL => if Substring.size (l) <= 1
                    then raise ConfigError
                  else Substring.string (Substring.trimr (l,1))
       | _ => raise ConfigError (* not right extension *)
  end

  fun simplerule (source) = let
    val name = stripCfgExt (source)
    fun default (f) = (f, NONE)
  in
    [default (result)]
  end

  val validator = stdTStampValidator

  fun processor = {source,targets} =
    (case targets
       of [] => raise ToolError {msg="huh?", tool=toolName}
        | (out,_)::_ => processFile (source,out)
                           handle e => raise ToolError
                                            {msg = exnMessage (e),
                                              tool=toolName})

  fun sfx (s) = Tools.addClassifier (Tools.stdSfxClassifier {sfx=s,class=class})

  val _ = Tools.addToolClass {class=class,
                              rule=Tools.dontcare simplerule,
                              validator=validator,
                              processor=processor}
  val _ = sfx ".cfg"

end
```

Figure 6.7: The structure *InstallMLConfig*

better way would be to specify in the description file an external tool to perform the processing[2].

As a matter of practice, and since we are already halfway there, let us transform the above into an external tool, that is a standalone tool that can be invoked from the operating system command line. Since in such a setting we will not have access to the interactive loop to set the various configuration variables, we will instead choose to read the settings from a file `.mlconfigrc` in the user's home directory[3]. A typical `.mlconfigrc` file would be:

```
name = Riccardo Pucella
date = August 17, 2000
```

Since we took pains to separate the core of the tool from the installation process, we can simply create a new CM description file for the external tool, `sources.cm`, which loads in `mlconfig.cm` implementing the core of the tool, as well as a file `driver.sml` to implement the main driver of the tool. Which turns out to be dead simple. Following our discussion in Section 5.4, we write a structure *Main* matching signature *MAIN* which provides the right interface for *SMLofNJ.exportFn* to create an executable.

```
structure Main = struct

  fun readRCFile (f) = ...

  fun main (_,[]) = (print "Need to specify a configuration file\n";
                     OS.Process.failure)
    | main (_,file::_) = (readRCFile ("~/.mlconfigrc");
                          MLConfig.processFile (file,MLConfig.stripCfgExt (file)) handle ...
                          OS.Process.success)
end
```

We can then compile the code through *CM.make ()* and export the tool through *SMLofNJ.exportFn ("mlconfig",Main.main)*. This creates a heap image `mlconfig.<something>` in the working directory.

Under Unix, we can create a simple shell script `ml-config` to invoke ML-Config:

```
#!/bin/sh
sml @SMLload=./mlconfig
```

We still need to register the tool class with CM. The installation code is just like that in Figure 6.7, except that the function *processor* now needs to invoke the external tool. We can therefore simply define:

[2]The latest version of CM (currently available through the working versions of SML/NJ) does indeed provide such a facility. This should make its way into the next release version, at which point these notes will be updated!

[3]This is the traditional Unix approach. Under Windows, such settings typically go into the Registry, but access to the Registry from within SML/NJ is still not available.

```
val person = CM.Tools.stdShellProcessor {command="ml-config",
                                          tool="ML-Config"}
```

Note that making the tool external does not really solve our earlier problem, that of requiring the user to explicitely install the tool class. But at least the processing can be performed off-line, without requiring CM intervention, so that purely SML-based code can be distributed.

6.5 Technicalities

Notes

The official documentation for CM is the user's manual available from the SML/NJ web page. The details and the theory underlying CM's approach to hierarchical modularity can be found in Blume's thesis [15] and papers [17, 16]. The kind of separate compilation found in CM is known as cutoff recompilation [2]: even if a source is modified, if the modification is found to have no effect on other files (even those formally "depending" on the modified files), the recompilation does not propagate any further. Implementation-wise, CM uses the hooks provided by the open compiler of SML/NJ [9].

CM is loosely based on SC, an early incremental compilation manager for SML/NJ [45, 46]. CM can still process SC description files, which are similar to CM description files but do not include the notion of subgroups or tools. The Unix program *make* was originally described in [28].

The configuration tool described in Section 6.4 is inspired by the configuration processing performed by *AutoConf* [**?**]. The RCS version control system was originally described in [106], and good references include [**?**].

One of the main roles of CM is in compiling the SML/NJ compiler itself. This is trickier than one might expect, as SML/NJ is itself written in SML (except for the runtime system, written in C). An eye-opening overview of the so-called bootstrap process can be found in [6].

Chapter 7

The SML/NJ Library

Back in Chapter 4, we studied the Basis Library, a standard set of modules that compliant SML'97 implementations should support. The focus of the Basis Library is on the support for basic types such as integers, words, strings, as well as common aggregate types such as lists, vectors and arrays. A large part of the Basis deals with system-independent functionality, such as filesystem access, input and output, and so on.

The SML/NJ Library is a set of modules distributed with SML/NJ to complete the Basis Library. It provides support for more specialized data structures, such as sets, maps and hash tables, with a variety of implementations. It also supports convenient formatted conversion from various types to strings, sorting functions, and generic higher-order functions. It also provides a module to handle command-line arguments in a sane way.

The focus in this chapter is on the so-called "utility" component of the SML/NJ Library. The "regular expressions" component of the SML/NJ Library will be described in Chapter **??**, while other components (HTML, PrettyPrinter, Reactive) are still under development and will be described in future versions of these notes.

7.1 Overview

To use any module from the SML/NJ library, you simply add the entry *smlnj-lib.cm* in the CM description file of your application. This does mean that projects using the SML/NJ Library will benefit greatly from using CM. I do not view this as a great restriction, since one of the goals of these notes is to promote the use of CM. I will reluctantly note that the SML/NJ Library is auto-loaded at toplevel by default in most installations (see Section **??**).

The SML/NJ Library uses the same conventions as the Basis Library with re-

```
structure LibBase : sig

  exception Unimplemented of string
  exception Impossible of string
  exception NotFound

  val failure : module : string, func : string, msg : string -> 'a
  val version : date : string, system : string, version_id : int list
  val banner : string

end
```

Figure 7.1: The structure *LibBase*

spect to names of structures, signatures, values, types and exceptions (see Section **??**). Unfortunately, since the modules in the library often come from different sources, there is no clear stylistic guidelines or naming conventions enforced across identifiers.

As I mentionned earlier, the SML/NJ Library provides functionality which can be divided into the following rather arbitrary categories: basic data structures, array operations, maps and sets, hash tables, sorting, output and string formatting, command-line options handling, and miscellaneous functionality.

A structure *LibBase* (see Figure 7.1) provides basic versioning information about the SML/NJ Library, and implements various core exceptions: *Unimplemented* is raised when an unimplemented feature is being used, *Impossible* is raised to report internal errors, and *NotFound* is raised by searching operations . The function *failure* raises a *Fail* exception with a standard error format. The values *version* and *banner* provide versioning information, respectively as a record and as a string.

7.2 Types and data structures

Similarly to the Basis Library, the SML/NJ Library provides structures for various types and data structures that can be useful in general applications. As the more common types have already been defined by the Basis, the types and data structures provided by the SML/NJ Library are necessarily more specialized.

The most important type defined by the SML/NJ Library is the type of atoms. An atom is a special kind of string that can be checked for equality very efficiently.[1] The downside is that there are no operations like concatenation on atoms. The

[1]Checking two normal strings for equality typically takes time proportional to the length of the shortest of the strings.

```
structure Atom : sig

  type atom

  val atom : string -> atom
  val atom' : substring -> atom
  val toString : atom -> string
  val sameAtom : (atom * atom) -> bool
  val compare : (atom * atom) -> order
  val hash : atom -> word

end
```

Figure 7.2: The structure *Atom*

```
signature CharMap : sig

  type 'a char_map

  val mkCharMap : default : 'a, bindings : (string * 'a) list -> 'a char_map
  val mapChr : 'a char_map -> char -> 'a
  val mapStrChr : 'a char_map -> (string * int) -> 'a

end
```

Figure 7.3: The structure *CharMap*

signature for the structure *Atom* is given in Figure 7.2. The structure defines an abstract type *atom*, along with constructors *atom* and *atom'*, converting respectively a string or a substring (of type *Substring.substring*) to an atom, and a function *toString* to convert an atom back to a string. The function *sameAtom* returns *true* if the atoms are the same. Two atoms are the same if their underlying strings are equal as strings (therefore, case matters when comparing atoms). The function *compare* is the standard comparison function. Note however that the ordering on atoms is not the lexicographic ordering on the underlying strings, but rather arbitrarily depends on the implementation of atoms. Finally, the function *hash* returns a word value that can be used to hash atoms. We will return to hash tables in Section **??**.

Another useful data structures is the character map, which maps characters to values. The signature of *CharMap* is given in Figure 7.3. It defines a type *'a char_map*, mapping characters (of type *Char.char*) to values of type *'a*. The function *mkCharMap* constructs a character map, by taking as argument a value of type $\{$*default:'a,bindings:(string* $\times$*'a) list*$\}$. The constructed character map, with the list *bindings* containing entries *(s,v)* of type *(string* $\times$*'a) list*, maps any

```
structure Fifo : sig

  type 'a fifo

  exception Dequeue

  val empty : 'a fifo
  val isEmpty : 'a fifo -> bool
  val enqueue : 'a fifo * 'a -> 'a fifo
  val dequeue : 'a fifo -> 'a fifo * 'a
  val delete : ('a fifo * ('a -> bool)) -> 'a fifo
  val head : 'a fifo -> 'a
  val peek : 'a fifo -> 'a option
  val length : 'a fifo -> int
  val contents : 'a fifo -> 'a list
  val app : ('a -> unit) -> 'a fifo -> unit
  val map : ('a -> 'b) -> 'a fifo -> 'b fifo
  val foldl : ('a * 'b -> 'b) -> 'b -> 'a fifo -> 'b
  val foldr : ('a * 'b -> 'b) -> 'b -> 'a fifo -> 'b

end
```

Figure 7.4: The structure *Fifo*

character in *s* into the value *v*. Characters not specified in any of the strings in *bindings* get mapped to the value specified by *default*. If a character is specified in multiple strings, the latest such binding (in the order of occurence of the strings in the list *bindings*) is dominant. The function *mapChr* takes a character map and a character and returns the value corresponding to according to the character map. The function *mapStrChr* simply applies *mapChr* to a character in the specified string. Thus, *mapStrChr cm (s,i)* is equivalent to *mapChr cm (String.sub (s,i))*.

We have seen in Chapter **??** various implementations of queues. The SML/NJ Library in fact provides such implementation, for both functional and imperative queues. Figure 7.4 gives the signature of *Fifo*, the structure implementing functional (or applicative) queues. I will not go over it in detail, since we have already discussed queues in Chapter **??**. The values and functions *empty*, *isEmpty*, *enqueue*, *head* and *dequeue* are as one expects. If the queue is empty, *head* and *dequeue* raise the *Dequeue* exception defined in *Fifo*. The alternative function *peek* also looks at the head of the queue, but returns an optional value: *SOME (v)* if *v* is at the head of the queue, *NONE* if the queue is empty. The function *length* returns the number of elements in the queue, while *contents* returns a list of the elements stored in the queue, in order from the head of the queue to the end. The remaining functions iterate over the elements of the queue: *delete* removes every element of the queue for which the supplied predicate returns *true*, whicle *app*, *map*, *foldl* and

```
structure Queue : sig

  type 'a queue

  exception Dequeue

  val mkQueue : unit -> 'a queue
  val clear : 'a queue -> unit
  val isEmpty : 'a queue -> bool
  val enqueue : 'a queue * 'a -> unit
  val dequeue : 'a queue -> 'a
  val delete : ('a queue * ('a -> bool)) -> unit
  val head : 'a queue -> 'a
  val peek : 'a queue -> 'a option
  val length : 'a queue -> int
  val contents : 'a queue -> 'a list
  val app : ('a -> unit) -> 'a queue -> unit
  val map : ('a -> 'b) -> 'a queue -> 'b queue
  val foldl : ('a * 'b -> 'b) -> 'b -> 'a queue -> 'b
  val foldr : ('a * 'b -> 'b) -> 'b -> 'a queue -> 'b

end
```

Figure 7.5: The structure *Queue*

foldr are the standard higher-order functions, acting on the queue from the head to the end.

Imperative queues are provided through the structure *Queue* whose signature is given in Figure 7.5. Aside from the fact that the functions act in-place on the queue provided (they do not return a new queue), most operations are just like for functional queues above, and we will not describe them again. The one difference is in constructing new queues. While *Fifo* provided a value *empty* defining the empty queue, *Queue* defines a function *mkQueue* to create a new empty queue, and a function *clear* to remove all the elements stored in a queue.

An interesting data structure defined by the SML/NJ Library is the splay tree, which is a form of balanced binary tree. The signature of *SplayTree* is given in Figure 7.6. It defines a datatype *'a splay* for splay trees, the typical definition one expects for binary trees. It also defines two operations that... . Splay trees are used internally by the SML/NJ Library to implement efficient sets and maps (see Section 7.4).

The next data structures we talk about in this section are property lists, which are not properly speaking data structures. A property list is an association list coupled to an object. A property list constains zero or more entries; each entry associates with a key (called an indicator) an arbitrary value (called the property)

```
structure SplayTree : sig

  datatype 'a splay =
      SplayObj of {
        value : 'a,
        right : 'a splay,
        left : 'a splay
      }
    | SplayNil

  val splay : (('a -> order) * 'a splay) -> (order * 'a splay)
  val join : 'a splay * 'a splay -> 'a splay

end
```

Figure 7.6: The structure *SplayTree*

```
structure PropList : sig

  type holder

  val newHolder : unit -> holder
  val clearHolder : holder -> unit
  val newProp : (('a -> holder) * ('a -> 'b)) -> {
            peekFn : 'a -> 'b option,
            getFn  : 'a -> 'b,
            clrFn  : 'a -> unit
          }
  val newFlag : ('a -> holder) -> {
            getFn : 'a -> bool,
            setFn : ('a * bool) -> unit
          }

end
```

Figure 7.7: The structure *PropList*

of a specific type. One can create new indicators and add them to any property list. The main advantage of a property list is that it can grow and can be used to associate values of different types to objects. Property lists are contained inside a holder. An object to which one wants to add properties needs to somehow define a holder. Typically an object will be a tuple or a record and the holder will be contained in a field. The structure *PropList* given in Figure 7.7 declares a type *holder* to hold the properties of a value. A new holder is created by calling *newHolder*.

The core of the work is done by *newProp*, which defines a new property of type *'b* for objects of some type *'a*. The function takes as arguments a function *'a →holder* that extracts the holder of the object, and a function *'a →'b* to create the initial value of the property. It returns a record of functions to access the property on objects of type *'a*. The returned function *getFn* takes the object and returns the property value associated to it; it creates the property for the object if it has not already been created, and initializes it using the initialization function. The returned function *peekFn* similarly gets at the property value, but returns it as an optional value, with *NONE* indicating that the property has not been created yet for that value. The returned function *clrFn* removes the property value, allowing it to be initialized at the next *getFn* call. Clearing all the properties in a holder is done by calling *clearHolder* on the holder.

A special kind of property, a boolean-value property type often called a flag, is given a special treatment through a *newFlag* function, since it is so common. A flag is automatically initialized to *false*, and thus does not require an initialization function. There is also no clear function in the record of functions returned by *mkFlag*. On the other hand, a returned function *setFn* can be used to change the setting of a flag.

The final data structures we discuss in this section are the union/find data structures, or ureferences (urefs). Ureferences are like normal references, except with the possibility of performing a union of the references. With normal references, the equality operating, checking so-called pointer equality, returns *true* if and only if two references cells are actually the same reference cell. Clearly this means that two such references must contain the same value. Ureferences support all the reference operations (creation, update, dereference) but in addition support an operation *union* that makes two distinct ureferences equal with respect to pointer equality of ureferences. Since the two ureferences may contain different values before the union, and since we would like to preserve the invariant that pointer-equal ureferences contain the same value (after all, they are supposed to represent the same cell after a union), we need to chose one of the values to store in the union.

Figure 7.8 presents the *UREF* signature, and two implementations, *SimpleUref* and *Uref*, with different degrees of cleverness in the implementation. They may exhibit different degrees of efficiency when multiple unions are performed. The

```
signature UREF = sig

  type 'a uref

  val uRef: 'a -> 'a uref
  val equal: 'a uref * 'a uref -> bool
  val !! : 'a uref -> 'a
  val update : 'a uref * 'a -> unit
  val unify : ('a * 'a -> 'a) -> 'a uref * 'a uref -> bool
  val union : 'a uref * 'a uref -> bool
  val link : 'a uref * 'a uref -> bool

end
```

Figure 7.8: The signature *UREF*

signature declares a type *'a uref* for ureferences, along with a constructor *uRef* to allocate a new ureference cell with a given initial value, a predicate *equal* to test pointer equality of ureferences, where *equal (e,e')* returns *true* if and only if *e* and *e'* have been unioned. The function *!!* returns the content of a ureference cell, while *update* updates its contents. (These functions correspond respectively to *!* and *:=* for references.) The unioning operation comes in three flavors: *union (e,e')* makes *e* and *e'* equal according to the *equal* predicate, and the content of the unioned cell is arbitrarily chosen to be one of the content of *e* or *e'*; the call *link (e,e')* does the same thing, expect that the content of the unified cell is explicitely taken to be the content of *e'* before the link; finally, *unify f (e,e')* applies a function *f* to the content of *e* and *e'* to compute the content of unioned cell. It should be clear that both *union* and *link* are expressible in terms of *unify*. In all cases, the unioning operation returns *true* if and only if the content of the ureferences were different before the union.

7.3 Arrays and vectors

The SML/NJ Library complements the support for arrays and vectors of the Basis Library (see Section 4.4). For instance, it provides a functor *MonoArrayFn* with the following declaration:

```
functor MonoArrayFn (type elem) :> MONO_ARRAY where type elem = elem
```

to construct new monomorphic array structures over an arbitrary element type. [2]

[2] The underlying implementation is in terms of polymorphic arrays, so there is no real efficiency gain.

```
structure BitVector : sig

  include MONO_VECTOR

  val fromString : string -> vector
  val bits : (int * int list) -> vector
  val getBits : vector -> int list
  val toString : vector -> string
  val isZero  : vector -> bool
  val extend0 : (vector * int) -> vector
  val extend1 : (vector * int) -> vector
  val eqBits : (vector * vector) -> bool
  val equal : (vector * vector) -> bool
  val andb : (vector * vector * int) -> vector
  val orb  : (vector * vector * int) -> vector
  val xorb : (vector * vector * int) -> vector
  val notb  : vector -> vector
  val lshift  : (vector * int) -> vector
  val rshift  : (vector * int) -> vector

end  where type elem = bool
```

Figure 7.9: The structure *BitVector*

A specialized implementation however is provided for bit vectors and bit arrays. As we did in Chapter 4, we focus initially on bit vectors, generalizing later to bit arrays. A bit vector is an efficient representation of vectors whose elements are single bits. The signature of structure *BitVector* is given in Figure 7.9, and is in fact an extension of the *MONO_VECTOR* signature (Figure **??**). We describe here the additional functionality of the structure, referring to Section **??** for the *MONO_VECTOR* operations.

Some of the additional operations are in charge of the conversion to and from bit vectors. The function *fromString* creates a new bit vector from a string argument giving the hexadecimal representation of the content of the bit vector. Each hexadecimal digit gives a 4-bit pattern in the vector, corresponding to the binary expansion of the digit (as usual, 1 corresponds to setting the corresponding bit in the vector). An exception *LibBase.BadArg* is raised if the string contains a non-hexadecimal character. The resulting vector always has a length of 4 times the length of the string. For example, calling *BitVector.fromString ("0F0F")* will create a bit vector of length 16 countaining the bits *0000111100001111*. The function *toString* preforms the exact inverse operation, creating an hexadecimal representation of the content of the bit vector. Note that the bit vector is zero-padded to a length which is a multiple of 4, prior to the conversion.

An alternative way of creating bit vectors is to use the function *bits*, which takes a length and a list of indices (as usual, indices are between 0 and *length-1* inclusively), and creates a new vector of the specified length with the bits at the given indices set. The exception *Subscript* is raised if an index in the list is out of range. The function *getBits* is the inverse operation, returning the list of indices corresponding to the set bits of a vector, in increasing order.

The remaining operations allow for specialized handling of bit vectors. The function *extend0* and *extend1* take a bit vector and a length and return a new vector which is the original vector extended on the right to the given length by either 0's or 1's. If the original vector is already longer than the prescribed length, it is returned unchanged. An exception *Size* is raised if the length is less than zero.

The predicate *isZero* tests if no bits are sets in a bit vector, while *eqBits* tests whether two vectors have the same bits set (that is, that the indices of the set bits are the same). Two vectors may *eqBits* to *true* while having different lengths. On the other hand, the function *equal* tests that the same bits are set and also that the lengths are the same.

The functions *andb*, *orb* and *xorb* take two bit vectors *a*,*b* and a length *l* and create a new bit vector of length *l*, with every element of the vector the AND (respectively OR and XOR, see page **??**) of the corresponding elements of *a* and *b*. If necessary, *a* and *b* are extended on the right with 0's. The function *notb* takes a bit vector and creates a new bit vector with all the bits of the supplied bit vector inverted.

The shift functions are slightly counter-intuitive on bit vectors, as opposed say to their *Word* counterparts, because the interpretation of bit vectors as numbers is not as intuitive. Nevertheless, the *lshift* function takes a bit vector and an integer *n* and creates a new vector by inserting *n* 0's on the right of the input vector. The function *rshift* takes a bit vector and an integer *n* and creates a new vector by removing *n* bits from the left of the input vector. If *n* is greater than or equal to the length of the input vector, the resulting vector has length 0.

The structure *BitArray* has a signature given in Figure 7.10. As expected, the signature is an extension of the *MONO_ARRAY* signature (Figure **??**). The majority of the additional functions behave for bit arrays as they do for bit vectors, so we will not cover them again. Operations specific to bit arrays include *setBit* and *clrBit*, to set or reset a given bit in the array (equivalent to the appropriate update operation provided by the *MONO_ARRAY* signature). The functions *union* and *intersection* take two bit arrays *a* and *b* and perform an OR (respectively AND) of the elements of *a* and *b* back into the array *a*, conceptually truncating or extending *b* on the right with 0's as needed. The function *complement* inverts the bits of an array, in place.

Bit vectors and bit arrays form an important part of the support for vectors and arrays in the SML/NJ Library. The second important part is the support for so-

```
structure BitArray : sig

  include MONO_ARRAY

  val fromString : string -> array
  val bits : (int * int list) -> array
  val getBits : array -> int list
  val toString : array -> string
  val isZero  : array -> bool
  val extend0 : (array * int) -> array
  val extend1 : (array * int) -> array
  val eqBits : (array * array) -> bool
  val equal : (array * array) -> bool
  val andb : (array * array * int) -> array
  val orb  : (array * array * int) -> array
  val xorb : (array * array * int) -> array
  val notb  : array -> array
  val lshift  : (array * int) -> array
  val rshift  : (array * int) -> array
  val setBit : (array * int) -> unit
  val clrBit : (array * int) -> unit
  val union : array -> array -> unit
  val intersection : array -> array -> unit
  val complement : array -> unit

end  where type elem = bool
```

Figure 7.10: The structure *BitArray*

```
structure DynamicArray : sig

  type 'a array

  val array : (int * 'a) -> 'a array
  val subArray : ('a array * int * int) -> 'a array
  val fromList : 'a list * 'a -> 'a array
  val tabulate: (int * (int -> 'a) * 'a) -> 'a array
  val default : 'a array -> 'a
  val sub : ('a array * int) -> 'a
  val update : ('a array * int * 'a) -> unit
  val bound : 'a array -> int
  val truncate : ('a array * int) -> unit

end
```

Figure 7.11: The structure *DynamicArray*

called dynamic arrays, which are arrays of unbounded length. As is the case for arrays, they come in two flavors: monomorphic and polymorphic. The operations provided are the same, only the types are different, with the kind type of differences encountered with arrays in the Basis Library. The structure *DynamicArray* implements polymorphic dynamic arrays, with the signature given in Figure 7.11. Since unbounded arrays cannot be effectively represented on a finite memory system, we achieve this effect by specifying a default value for the arrays. By default, querying an array position for its element returns the default value, unless that element has been explicitely updated by the program. Therefore, we conceptually only need to keep track of which array positions contain values that have been explicitely set, of which there can only be a finite number at any given time. One consequence of this fact that every operation which creates an array needs as an argument a default value for its unassigned elements, or at least needs to specify which default value it uses.

The structure *DynamicArray* declares a type *'a array* (incompatible with the *Array.array* type), and a function *array* taking an integer *n* and a default value *v*, and creates a new unbounded array whose elements are all initialized to *v*. The integer *n* is used as a hint to the potential useful range of indices.[3] The function *subArray* creates an array from the subrange of a given array, with the same default value as the given array, while *fromList* creates an array from a list of elements and a default value, and *tabulate* creates an array from a tabulating function for a given number of elements, with the rest of the elements given a default value.

The default value of an array can be queried for by the function *default*. The function *sub* looks at a given index in the array, and returns either the last value stored at that index, or the default value. Storing a new value in an array is done by the function *update*, as in the case of normal arrays. Specific to dynamic arrays are the functions *bound* which returns (an upperbound on) the largest index of any value that has been updated in the array, while *truncate* makes every elements in the array at an index larger than the supplied integer the default value, effectively truncating the array.

Monomorphic dynamic arrays are obtained from monomorphic arrays (of signature *MONO_ARRAY*) by an application of the functor *DynamicArrayFn*:

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

The signature of the resulting monomorphic dynamic array structure is similar to that of polymorphic arrays, except that the type of the elements is fixed to that specified by the underlying monomorphic array structure.

[3]This is purely for implementation efficiency pruposes.

Note that dynamic arrays do not provide iterators such as folding or map operations. On the other hand, in a completely call-by-value language, such operations are slightly nonsensical on unbounded arrays.

7.4 Sets and maps

The next set of data structures we consider include ordered sets and ordered maps. Ordered sets are data structures that aim at supporting set operations over values of an ordered type. Sets have the property that they do not allow repeated elements: adding an item to a set which already contains that item does not change the set. Furthermore, sets are unordered collections. The restriction to values of ordered types is to allow efficient implementation of sets; the ordering is not reflected at the level of the interface with sets. Ordered maps are similar in spirit. They associate with values of a given ordered type (the key) a value of some other type, thereby defining a mapping from keys to values.

Many programs benefit from an efficient implementation of sets and maps. Many programs not aware of the presence of such libraries end up "reinventing the wheel", or using simple lists or association lists to achieve the effects of sets and maps, at the cost of engineering efforts and lack of efficiency: not many programmers will go to the trouble of implementing red-black trees to get efficient sets and maps!

Both sets and maps are implemented as functors parameterized over the key type. For sets, this is the type of the values that can be stored in a set, while for maps it is the type of the keys to which values are associated. Different functors are available for both sets and maps, providing different implementations, each offering its own pros and cons. It is not the purpose of these notes to discuss in detail the algorithmic properties of these implementations. I will instead direct you to the notes at the end of this chapter, which provide references to such details.

As I mentionned, functors for building sets and maps are parameterized over the key type, via the following signature:

```
signature ORD_KEY = sig
  type ord_key
  val compare : ord_key * ord_key -> order
end
```

where once again our generic comparisong function appears. Any type for which a suitable *compare* function can be defined can be used as a key for sets and maps. For example, strings can be turned into keys as simply as:

```
structure StringKey = struct
  type ord_key = string
  val compare = String.compare
end
```

```
signature ORD_KEY = sig

  structure Key : ORD_KEY

  type item = Key.ord_key
  type set

  val empty : set
  val singleton : item -> set
  val add  : set * item -> set
  val add' : (item * set) -> set
  val addList : set * item list -> set
  val delete : set * item -> set
  val member : set * item -> bool
  val isEmpty : set -> bool
  val equal : (set * set) -> bool
  val compare : (set * set) -> order
  val isSubset : (set * set) -> bool
  val numItems : set ->  int
  val listItems : set -> item list
  val union : set * set -> set
  val intersection : set * set -> set
  val difference : set * set -> set
  val map : (item -> item) -> set -> set
  val app : (item -> unit) -> set -> unit
  val foldl : (item * 'b -> 'b) -> 'b -> set -> 'b
  val foldr : (item * 'b -> 'b) -> 'b -> set -> 'b
  val filter : (item -> bool) -> set -> set
  val exists : (item -> bool) -> set -> bool
  val find : (item -> bool) -> set -> item option

end
```

Figure 7.12: The signature *ORD_SET*

Fundamentally, both sets and maps are similar, and whatever implementation works for sets can be used for maps, and vice versa. Thus, the different implementations for sets are reflected in maps, in a natural way.

The SML/NJ Library provides four different implementations of sets and maps: one based on a class of binary search trees, one on sorted lists, one on splay trees, and one on red-black trees. Overall, the red-black trees implementation is the most efficient.

The signature *ORD_SET* for sets is given in Figure 7.12 and four functors (parameterized over an *ORD_KEY* signature) create matching structures (note that these functors use an unnamed parameter approach):

```
functor BinarySetFn (K:ORD_KEY):ORD_SET
functor ListSetFn (K:ORD_KEY):ORD_SET
functor SplaySetFn (K:ORD_KEY):ORD_SET
functor RedBlackSetFn (K:ORD_KEY):ORD_SET
```

The *ORD_SET* signature implemented by these functors provide the functions you would expect to find in any implementation of sets, and many others. It defines a type for both a set and its elements (which is the same type as *ord_key* from the parameter structure to the functor). A value *empty* represents the empty set. The function *singleton* creates a single element set, while *add* (and *add'*) add an element to a set,[4] and *addList* adds every element of a list to a set. The function *delete* deletes an element from a set, raising *LibBase.NotFound* if it is not found in the set. Note that the *compare* function provided by the functor parameter specifies when two elements are the same (namely when it returns *EQUAL*). The predicate *member* checks if an element is part of a set, *isEmpty* if the set is empty, and *equal* if two sets are equal. By the principle of extensionality, two sets are equal is they contain the same elements. The function *compare* compares two sets according to the lexicographic order of its elements, themselves ordered based on the order specified by the functor parameter. The function *isSubset* checks if the first set is a subset of the second, i.e. that every element in the first set is a member of the second. The functions *numItems* and *listItems* respectively return the number of items in a set and an ordered list of the elements of a set.

Traditional set operations such as *union*, *intersection* and *difference* are available. The remaining functions are the standard higher-order functions to manage and iterate over sets: *map*, *app*, *foldl*, *foldr*. Note that *map*, *app* and *foldl* walk the elements of the sets in increasing order, while *foldr* walks the set in decreasing order. The function *filter* returns a new set, the subset of the initial set made up of all the elements for which the supplied predicate evaluates to *true*, while *exists* merely checks that a given element exists satisfying the predicate. The function *find* walks the set in increasing order and returns *SOME (v)* for *v* the first element for which the supplied predicate evaluates to *true*, or *NONE* if no such element exists.

It is often the case that one needs to extract some element of a set, arbitrarily, and efficiently. Such a function, often called *choose*, can be derived from the above functions as follows:

```
val choose = find (fn _ => true)
```

As an example, consider implementing sets of strings, as red-black trees. This can be done simply by:

```
structure StringSet = RedBlackSetFn (struct
                                       type ord_key = string
                                       val compare = String.compare
                                     end)
```

[4]These two functions are defined with only the order of the arguments as a difference. The reason for such a strange duplication is that the function *add* is often used as an argument to a fold, and implementations of fold sometimes differ as to the order of the arguments they pass.

Because of their widespread use, structures implementing sets of integers and sets of atoms are predefined in the library. For integers, the structures *IntBinarySet*, *IntListSet*, *IntRedBlackSet* are provided, while for atoms, *AtomBinarySet* and *AtomRedBlackSet* are provided. The structure *AtomSet* is a synonym for *AtomRedBlackSet*.

The signature *ORD_MAP* for maps is given in Figure 7.13, and just as in the case of sets, four functors (parameterized over an *ORD_KEY* signature) can be used to create a matching structure, one for each underlying implementation (see page):

```
functor BinaryMapFn (K:ORD_KEY):ORD_SET
functor ListMapFn (K:ORD_KEY):ORD_SET
functor SplayMapFn (K:ORD_KEY):ORD_SET
functor RedBlackMapFn (K:ORD_KEY):ORD_SET
```

The *ORD_MAP* signature implemented by these functors defines a type for both maps and its keys (the type *ord_key* inherited from the functor parameter). The value *empty* represents the empty map, which can be checked for by the predicate *isEmpty*. The function *singleton* creates a new map containing a single key and its associated value, while *insert* (and *insert'*) insert a new key and associated value in the map. (See the remarks on the functions *add* and *add'* from *ORD_SET* for a rationale.) To query a map for the values it contains, one can use the function *find*, which returns an optional value associated with a given key, or *NONE* if the key is not found in the map. The predicate *isDomain* checks whether a key is in the map. The function *remove* removes a key (and its associated value) from a map, raising *LibBase.NotFound* if the key is not in the map. The function *first* (respectively *firsti*) returns the first value stored in the map, or *NONE* if the map is empty (respectively, the first value and its associated key).

The function *numItems* returns the number of values stored in the map. The function *listItems* (respectively *listItemsi*) returns an ordered list of the values in the map (respectively an ordered list of the values and their associated key), ordered by the associated key. Similarly, *listKeys* returns an ordered list of the keys in the map.

The function *collate*, as the name indicates, constructs an ordering between maps, given an ordering on the values stored in the map. .

The operations *unionWith* and *intersectWith* are used to somehow merge two maps. The function *unionWith* (respectively *unionWithi*) returns a map whose domain is the union of the domains of the two supplied maps. For values whose keys are in both maps, a way to pick the value to put in the new map has to be devised. The solution is to pass a function taking as arguments the two values corresponding to the same key (respectively, the two values and the common key) and returning a new value. For example, to perform a map union giving precedence to

```
signature ORD_MAP = sig

  structure Key : ORD_KEY

  type 'a map

  val empty : 'a map
  val isEmpty : 'a map -> bool
  val singleton : (Key.ord_key * 'a) -> 'a map
  val insert  : 'a map * Key.ord_key * 'a -> 'a map
  val insert' : ((Key.ord_key * 'a) * 'a map) -> 'a map
  val find : 'a map * Key.ord_key -> 'a option
  val inDomain : ('a map * Key.ord_key) -> bool
  val remove : 'a map * Key.ord_key -> 'a map * 'a
  val first : 'a map -> 'a option
  val firsti : 'a map -> (Key.ord_key * 'a) option
  val numItems : 'a map ->  int
  val listItems  : 'a map -> 'a list
  val listItemsi : 'a map -> (Key.ord_key * 'a) list
  val listKeys : 'a map -> Key.ord_key list
  val collate : ('a * 'a -> order) -> ('a map * 'a map) -> order
  val unionWith  : ('a * 'a -> 'a) -> ('a map * 'a map) -> 'a map
  val unionWithi : (Key.ord_key * 'a * 'a -> 'a) -> ('a map * 'a map) -> 'a map
  val intersectWith  : ('a * 'b -> 'c) -> ('a map * 'b map) -> 'c map
  val intersectWithi : (Key.ord_key * 'a * 'b -> 'c) -> ('a map * 'b map) -> 'c map
  val app  : ('a -> unit) -> 'a map -> unit
  val appi : ((Key.ord_key * 'a) -> unit) -> 'a map -> unit
  val map  : ('a -> 'b) -> 'a map -> 'b map
  val mapi : (Key.ord_key * 'a -> 'b) -> 'a map -> 'b map
  val foldl  : ('a * 'b -> 'b) -> 'b -> 'a map -> 'b
  val foldli : (Key.ord_key * 'a * 'b -> 'b) -> 'b -> 'a map -> 'b
  val foldr  : ('a * 'b -> 'b) -> 'b -> 'a map -> 'b
  val foldri : (Key.ord_key * 'a * 'b -> 'b) -> 'b -> 'a map -> 'b
  val filter  : ('a -> bool) -> 'a map -> 'a map
  val filteri : (Key.ord_key * 'a -> bool) -> 'a map -> 'a map
  val mapPartial  : ('a -> 'b option) -> 'a map -> 'b map
  val mapPartiali : (Key.ord_key * 'a -> 'b option) -> 'a map -> 'b map

end
```

Figure 7.13: The signature *ORD_MAP*

the rightmost map, you could use:

```
unionWith (fn (x,y) => y) (map1,map2)
```

The operation *intersectWith* (respectively *intersectWithi*) creates a new map which contains keys appearing in both the supplied maps. Again, we need to pass in a function taking as arguments the two values associated with a common key (respectively, the two values and the common key) and returning a value to associate with the key in the new map.

The remaining functions, just as in the case of sets, are the strandard higher-order iteration functions. These functions operate on the values stored in the map, leaving the keys for the better part alone. Each function comes with an alternate version with an *i* appended to the name, indicating that not only the value is to be passed to any higher-order function, but the associated key as well. The type of the respective functions should make this clear. The functions *map*, *app*, *foldl*, *foldr* and *filter* perform the expected operations, all in increasing order on the associated keys (except for *foldr* which acts in decreasing order on the keys). The function *mapPartial* maps a partial function (see page **??**) over the elements of a map: if the result of the partial function is *SOME (v)*, then *v* is the value associated with the key in the new map, while a result of *NONE* indicates that the key and value should not be added to the new map. In effect, *mapPartial f m* is equivalent to:

```
map (Option.valOf (filter Option.isSome (map f m)))
```

Just as with sets, the SML/NJ Library predifines structures implementing maps for integer and atom keys. For integers, the structures *IntBinaryMap*, *IntListMap* and *IntRedBlackMap* are provided, while for atoms, *AtomBinaryMap* and *AtomRedBlackMap* are provided. As before, *AtomMap* is a synonym for *AtomRedBlackMap*.

7.5 Hash tables

Hashing can best be understood as a particularly efficient implementation of the map data structure. Recall from the previous section that a map associates with every key (of a given specified key type) a value. In general, for the implementations seen in the previous section, looking up the value associated with a given key takes at worst time logarithmic in the size of the structure.

A hash table attempts to provide a map structure supporting lookup operations that take constant time. This can be achieved by basically storing the map in an array, and using a hash function *h* that for each key returns the index in the array where the associated value is stored. In general, depending on the hash function,

```
signature HASH_KEY = sig
  type hash_key

  val hashVal : hash_key -> word
  val sameKey : (hash_key * hash_key) -> bool
end
```

Figure 7.14: The signature *HASH_KEY*

```
structure HashString : sig
  val hashString : string -> word
end
```

Figure 7.15: The structure *HashString*

there can be more than two keys for which the hash function returns the same index. A strategy for conflict resolution is needed for such cases. Rather than go into the details here, I will point you to the notes for this chapter, for references and further comments.

In the previous section, we saw that maps and sets were parameterized over *ORD_KEY*, a signature specifying a type for keys with an associated comparison function. A similar parameterization is used for hash tables, except that the parameterization is slightly different. The signature *HASH_KEY* specifies the type of keys used for a particular hash table, and is given in Figure 7.14. It defines a type *hash_key*, as a well as a function *hashVal* (the hash function), which returns an unsigned integer, and a function *sameKey* checking if two keys are equal. A general comparison function is not needed.

A hash function can be anything that returns an unsigned integer, but a hash table is more effective (namely, lookup time is kept at a minimum) when the function is as close to one-to-one as possible. Since strings are often used as keys, a hash function for strings is provided by the SML/NJ Library. Structure *HashString* (signature in Figure 7.15) implements a function *hashString* to compute a simple but effective hash value. The hash value accumulated character by character, using the formula:

$$h = 33h + 720 + c$$

Also note that atoms come with a built-in hash function (the function *Atom.hash*).

Hash tables come in two forms, like vectors and arrays: polymorphic and monomorphic. (Note that maps only come in the polymorphic variety.) The polymorphic hash tables structure can create hash tables for an arbitrary key type, while

```
structure HashTable : sig
  type ('a, 'b) hash_table

  val mkTable : (('a -> word) * (('a * 'a) -> bool)) -> (int * exn)
                  -> ('a,'b) hash_table
  val clear : ('a, 'b) hash_table -> unit
  val insert : ('a, 'b) hash_table -> ('a * 'b) -> unit
  val lookup : ('a, 'b) hash_table -> 'a -> 'b
  val find : ('a, 'b) hash_table -> 'a -> 'b option
  val remove : ('a, 'b) hash_table -> 'a -> 'b
  val numItems : ('a, 'b) hash_table ->  int
  val listItems  : ('a, 'b) hash_table -> 'b list
  val listItemsi : ('a, 'b) hash_table -> ('a * 'b) list
  val app  : ('b -> unit) -> ('a, 'b) hash_table -> unit
  val appi : (('a * 'b) -> unit) -> ('a, 'b) hash_table -> unit
  val map  : ('b -> 'c) -> ('a, 'b) hash_table -> ('a, 'c) hash_table
  val mapi : (('a * 'b) -> 'c) -> ('a, 'b) hash_table -> ('a, 'c) hash_table
  val fold  : (('b *'c) -> 'c) -> 'c -> ('a, 'b) hash_table -> 'c
  val foldi : (('a * 'b * 'c) -> 'c) -> 'c -> ('a, 'b) hash_table -> 'c
  val filter  : ('b -> bool) -> ('a, 'b) hash_table -> unit
  val filteri : (('a * 'b) -> bool) -> ('a, 'b) hash_table -> unit
  val copy : ('a, 'b) hash_table -> ('a, 'b) hash_table
  val bucketSizes : ('a, 'b) hash_table -> int list
end
```

Figure 7.16: The structure *HashTable*

monomorphic hash tables are created by a functor taking as argument a structure matching *HASH_KEY* representing the key type.

We discuss polymorphic hash tables first, as they are more general. Most of our descriptions will carry over verbatim to the monomorphic case. The structure *HashTable* (whose signature is given in Figure 7.16) implements polymorphic hash tables. The type of a hash table is *('a,'b) hash_table*, mapping keys of type *'a* to values of type *'b*.

The core operations on hash tables include *mkTable*, which creates a new hash table. Note that hash tables are fundamentally imperative structures. Because these are polymorphic hash tables, and that they are not restricted to one specific key type, we must supply both the hash function and the key equality operation for the type of keys we want to use on the particular hash table we are creating. The other arguments supplied to *mkTable* are a hint as to the size of the hash table (a hash table that is too large will waste space, while a hash table that is too small will waste time growing as more elements are added), and an exception to be raised by *lookup* and *remove*. We refer to this exception as the table's exception.

Adding entries to the table is done through *insert*, which takes the table into which to insert, and the pair *(key,value)* to enter in the table. The table is changed as a side-effect. Looking up values associated with keys can be done in two ways, just like for the maps in Section **??**. The function *lookup* searches for the value associated with a given key, and returns that value if it exists; it raises the table's exception if the key is not associated to any value. The function *find* is similar, except that it returns an option value: *SOME (v)* if *v* is associated with the given key, *NONE* otherwise. Which version to use is largely a matter of preference To remove associations from the hash table, you can either use *remove*, which removes the value associated with a given key, returning that value (raising the table's exception if the key has no association), or use *clear* which removes all the associations from a hash table.

The remaining functions follow the definitions of the map operations, as hash tables are just maps. Thus, *numItems* returns the number of values stored in the hash table, while *listItems* (respectively *listItemsi*) returns a list of the values (respectively, pairs of key and value) in the table, in some arbitrary order. The iteration functions *map*, *app*, *fold*, *filter* and associated operations *mapi*, *appi*, *foldi* and *filteri* (which act on both keys and values) are as for general maps.

Two small differences: the *HashTable* structure provides a function *copy* to create a copy of the hash table (this is sometimes needed as hash tables are imperative structures.) You cannot really write such a function yourself, as it is not possible to get at the hash function and equality predicate of a hash table. Also, the function *bucketSizes* returns a list of the sizes of the various buckets This can be useful to gauge the quality of the hash function: a good hash function should keep the size of the buckets approximately the same.

Monomorphic hash tables are similar to polymorphic hash tables, except that the type of the keys is fixed. A functor is used to build a monomorphic hash table given a structure matching signature *HASH_KEY*, with the following declaration:

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

The structure obtained by applying the functor matches the signature *MONO_HASH_TABLE* given in Figure 7.17. The functionality implemented by the structure is virtually identical as that implemented by the *HashTable* structure for polymorphic hash tables. Differences include: the type of hash tables is simply *'a hash_table*, where *'a* is the type of stored values (the type of the key need not be mentioned since it is fixed by the structure), the constructor function *mkTable* only takes a size hint and the table's exception (the equality operation on keys, as well as the hash function, is fixed by the structure). Moreover, the key information used by the structure is available in a substructure *Key*.

A variation on hash tables is available, namely monomorphic hash tables indexed by two keys. An item is inserted under two keys, and can be retrieved by either key. Just as in the standard monomorphic hash tables case, a functor is used

```
signature MONO_HASH_TABLE = sig
  structure Key : HASH_KEY

  type 'a hash_table

  val mkTable : (int * exn) -> 'a hash_table
  val clear : 'a hash_table -> unit
  val insert : 'a hash_table -> (Key.hash_key * 'a) -> unit
  val lookup : 'a hash_table -> Key.hash_key -> 'a
  val find : 'a hash_table -> Key.hash_key -> 'a option
  val remove : 'a hash_table -> Key.hash_key -> 'a
  val numItems : 'a hash_table ->  int
  val listItems  : 'a hash_table -> 'a list
  val listItemsi : 'a hash_table -> (Key.hash_key * 'a) list
  val app  : ('a -> unit) -> 'a hash_table -> unit
  val appi : ((Key.hash_key * 'a) -> unit) -> 'a hash_table -> unit
  val map  : ('a -> 'b) -> 'a hash_table -> 'b hash_table
  val mapi : ((Key.hash_key * 'a) -> 'b) -> 'a hash_table -> 'b hash_table
  val fold  : (('a * 'b) -> 'b) -> 'b -> 'a hash_table -> 'b
  val foldi : ((Key.hash_key * 'a * 'b) -> 'b) -> 'b -> 'a hash_table -> 'b
  val filter  : ('a -> bool) -> 'a hash_table -> unit
  val filteri : ((Key.hash_key * 'a) -> bool) -> 'a hash_table -> unit
  val copy : 'a hash_table -> 'a hash_table
  val bucketSizes : 'a hash_table -> int list
end
```

Figure 7.17: The signature *MONO_HASH_TABLE*

```
signature MONO_HASH2_TABLE = sig
  structure Key1 : HASH_KEY
  structure Key2 : HASH_KEY

  type 'a hash_table

  val mkTable : (int * exn) -> 'a hash_table
  val clear : 'a hash_table -> unit
  val insert : 'a hash_table -> (Key1.hash_key * Key2.hash_key * 'a) -> unit
  val lookup1 : 'a hash_table -> Key1.hash_key -> 'a
  val lookup2 : 'a hash_table -> Key2.hash_key -> 'a
  val find1 : 'a hash_table -> Key1.hash_key -> 'a option
  val find2 : 'a hash_table -> Key2.hash_key -> 'a option
  val remove1 : 'a hash_table -> Key1.hash_key -> 'a
  val remove2 : 'a hash_table -> Key2.hash_key -> 'a
  val numItems : 'a hash_table ->  int
  val listItems  : 'a hash_table -> 'a list
  val listItemsi : 'a hash_table -> (Key1.hash_key * Key2.hash_key * 'a) list
  val app  : ('a -> unit) -> 'a hash_table -> unit
  val appi : ((Key1.hash_key * Key2.hash_key * 'a) -> unit) -> 'a hash_table
                   -> unit
  val map  : ('a -> 'b) -> 'a hash_table -> 'b hash_table
  val mapi : ((Key1.hash_key * Key2.hash_key * 'a) -> 'b) -> 'a hash_table
                   -> 'b hash_table
  val fold  : (('a * 'b) -> 'b) -> 'b -> 'a hash_table -> 'b
  val foldi : ((Key1.hash_key * Key2.hash_key * 'a * 'b) -> 'b) -> 'b
                   -> 'a hash_table -> 'b
  val filter  : ('a -> bool) -> 'a hash_table -> unit
  val filteri : ((Key1.hash_key * Key2.hash_key * 'a) -> bool) -> 'a hash_table
                   -> unit
  val copy : 'a hash_table -> 'a hash_table
  val bucketSizes : 'a hash_table -> (int list * int list)
end
```

Figure 7.18: The signature *MONO_HASH2_TABLE*

to construct an implementation, taking as parameters two structures matching the signature *HASH_KEY*. The functor is declared as follows:

```
functor Hash2TableFn (structure Key1 : HASH_KEY
                      structure Key2 : HASH_KEY): MONO_HASH2_TABLE
```

The structure obtained by applying the functor matches signature *MONO_HASH2_TABLE* given in Figure 7.18. The functionality implemented is just as for standard monomorphic hash tables, except that most functions expecting a key now expect two keys. The *lookup*, *find* and *remove* functions are not available, but there are functions *lookup1* and *lookup2* that perform a lookup operation on the first key and the second key respectively (similarly for *find1*, *find2*, *remove1* and *remove2*). Note that the semantics for removal affects insertion as well: if an item is inserted at keys k_1,k_2 and an item already is associated with either k_1 or k_2, that item is removed

```
signature LIST_SORT = sig

  val sort : ('a * 'a -> bool) -> 'a list -> 'a list
  val uniqueSort : ('a * 'a -> order) -> 'a list -> 'a list
  val sorted : ('a * 'a -> bool) -> 'a list -> bool

end
```

Figure 7.19: The signature *LIST_SORT*

prior to insertion.

As a final remark on hash tables, note that in keeping with maps and sets, a structure *AtomTable* (matching *MONO_HASH_TABLE*) is already instantiated in the SML/NJ Library, implementing hash tables indexed by atoms.

7.6 Sorting

Sorting is the quintessential example of functionality that belongs in a library. The SML/NJ Library provides basic facilities for sorting lists (based on the MergeSort algorithm), and for sorting arrays (based on the QuickSort algorithm).

Sorting lists is supported through a signature *LIST_SORT*, that sorting structures should implement. The signature is given in Figure 7.19. At the present time, only one structure in the library implements *LIST_SORT*, namely *ListMergeSort*, based on the MergeSort algorithm.

The signature specifies a function *sort* taking a boolean-valued comparison operation as an argument, as well as a list, and sorts the list using the given comparison function. It returns the sorted list. For a comparison function *c*, the resulting list *[*x_1,...,x_n*]* satisfies: if *c (*x_i,x_j*)* evaluates to *true*, then $i > j$. Note that we do not specify what happens when two values are deemed equal by the comparison function.

The function *sorted* takes a comparison operator and a list as arguments, and checks whether the list is sorted according to the definition of a sorted list given in the previous paragraph: a list *[*x_1,...,x_n*]* is sorted relative to the comparison operation *c* if when *c (*x_i,x_j*)* evaluates to *true* then $i > j$.

The function *uniqueSort* takes an order-valued comparison operation (that is, of type *'a tuple 'a →order* and a list and produces a sorted list without repetition of equal elements. That is, if an element is equal to another element (according to the comparison operation), only one such element is kept. (Which element is kept is arbitrary.) The resulting list contains no duplicates, and is sorted according to the definition of the previous paragraph: *[x_1,...,x_n]* is sorted relative to the comparison operator *c* if when *c* (x_i,x_j) evaluates to *GREATER*, then $i > j$. The fact that the resulting list contains no duplicates can be expressed by the property: if *c* (x_i,x_j) evaluates to *EQUAL*, then $i = j$.

Consider the following examples. Sorting a list of integers *L* in increasing order is done simply by:

```
ListMergeSort.sort (Int.>) L
```

Sorting *L* in decreasing order is similarly simple:

```
ListMergeSort.sort (Int.<) L
```

Sorting *L* in increasing order according to the last digit of the number is done by the following expression. Note that ordering by last digit only defines a partial order, so a number of resulting lists are considered sorted. The algorithm picks one.

```
ListMergeSort.sort (fn (x,y) => (x mod 10) > (y mod 10)) L
```

If *L* is *[46,37,16,8,32,15,20]*, the result of the above evaluations are respectively *[8,15,16,20,32,37,46]*, *[46,37,32,20,16,15,8]* and *[20,32,15,46,16,37,8]*.

The functionality of *uniqueSort* relies heavily on the definition of the comparison operation. Going back to the sorting-according-to-the-last-digit example, we can easily sort and eliminate duplicate numbers by:

```
ListMergeSort.uniqueSort (fn (x,y) => let
                            val x' = x mod 10
                            val y' = y mod 10
                          in
                            if (x=y) then EQUAL
                            else if (x'>y') then GREATER
                            else LESS
                          end) L
```

However, one may also consider the equivalence classes of numbers with the same last digit, and decide to keep only one representative from each class, which leads to the following revised definition:

```
ListMergeSort.uniqueSort (fn (x,y) => Int.compare (x mod 10, y mod 10)) L
```

When *L* is *[15,5,3,20,43]*, the above expressions respectively yield *[20,43,3,5,15]* and *[20,43,5]*. Note that it is not possible to specify which value is kept when the comparison operation deems two values to be equal.

Whereas sorting a list produces a new list, sorting an array is done in-place.[5] Since arrays come in two flavors, monomorphic and polymorphic, the SML/NJ

```
signature ARRAY_SORT = sig

  type 'a array

  val sort   : ('a * 'a -> order) -> 'a array -> unit
  val sorted : ('a * 'a -> order) -> 'a array -> bool

end
```

Figure 7.20: The signature *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
```

Figure 7.21: The signature *MONO_ARRAY_SORT*

Library provides facilities for sorting both. We start by describing how to sort polymorphic arrays.

The signature *ARRAY_SORT* specifies the functionality of structures for sorting polymorphic arrays. The signature is given in Figure 7.20. It declares a function *sort* to sort an array in-place, given a comparison operation *c*. In contrast to list sorting, this comparison function is order-valued. The definition of a sorted array is similar to that given for lists above: an array *r* containing $x_1, \ldots, x_n$ is sorted relative to *c* if when *c* (x_i, x_j) evaluates to *GREATER*, then $i > j$. The function *sorted* simply checks if an array is sorted according to this definition. The SML/NJ Library currently provides a single structure matching *ARRAY_SORT*, the structure *ArrayQSort*, that implements an engineered version of QuickSort. Note that the signature defines a type *'a array*, that *ArrayQSort* transparently identifies with the type *Array.array*.

For sorting monomorphic arrays, the idea is similar, except that everything is achieved through functors. A signature *MONO_ARRAY_SORT* (given in Figure 7.21) specifies structures to sort a given kind of *MONO_ARRAY*. It provides the same fucntionality as the *ARRAY_SORT* signature of the previous paragraph. A

[5] One could sort vectors as well, which as in the case of lists would produce a new vector. This is because lists and vectors are immutable structures, as opposed to arrays.

```
functor BSearchFn (A : MONO_ARRAY) : sig
  structure A : MONO_ARRAY
  val bsearch : (('a * A.elem) -> order) -> ('a * A.array) -> (int * A.elem) option
end
```

Figure 7.22: The functor *BSearchFn*

functor *ArrayQSortFn* is provided by the Library, declared as follows:

```
functor ArrayQSortFn (A:MONO_ARRAY):MONO_ARRAY_SORT
```

that takes as parameter a structure implementing monomorphic arrays of some type, and that creates a structure for sorting those arrays, using the engineered QuickSort algorithm mentionned above.

There is one more facility associated with sorted monomorphic arrays, and that is the ability to perform a binary search. Recall that one can perform an efficient search in a sorted array by a divide-and-conquer strategy: say you are searching for an element x in the array. Compare x to the value m in the middle of the array: if $x = m$, you are done, if $x < m$, recursively search for x in the left-half of the array, otherwise recursively search for x in the right-half of the array. This leads to a worst-case search time that is logarithmic in the size of the array. The functor *BSearchFn* (given in Figure 7.22) produces a structure implementing a function to perform such a binary search on the monomorphic arrays provided as a parameter to the functor.

The *bsearch* function implemented by the resulting structure is in fact a polymorphic search function: it searches for values of type *'a*. How can this make sense if the array is monomorphic, and so stores a specific type of value? The answer is the comparison operation, the first argument to *bsearch*. The comparison operation has type *'a* * *A.elem* →*order* (where *A* is the monomorphic array structure used as a parameter to the functor), and thus performs a comparison between a generic value and the array element. It is up to the comparison function to extract a value of type *A.elem* from values of type *'a* to actually perform the comparison. The function *bsearch* also takes as arguments the value of type *'a* to look for, as well as the array to search. Note that the array must be sorted, relative to a comparison operation which agrees with the comparison operation passed to *bsearch*. The result of the search is an option value, *SOME (i,v)* if the value searched for is found at index *i* with stored value *v*, and *NONE* otherwise.

```
structure 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)
      | RIGHT of (int * fmt_item)

  exception BadFormat
  exception BadFmtList

  val format  : string -> fmt_item list -> string
  val formatf : string -> (string -> unit) -> fmt_item list -> unit

end
```

Figure 7.23: The structure *Format*

7.7 Formatting

The SML/NJ Library provides facilities for formatting output and reading formatted input. These facilities correspond to the ANSI C *sprintf* and *sscanf* functions. For formatting output, a structure *Format* (with signature given in Figure 7.23) declares a function *format* taking as input a format string, that is a string with embedded directives to be replaced by values of the actual type, as well as a list of formatting items denoting the values to insert in the format string. The result of the function is a new string with the directives (which are placeholders within the format string) replaced by a string representation of the actual values.

The formatting characters in the format string indicate the type of the value to insert at a given point. The formatting items indicate the value to insert in the resulting string, and is specified as a large datatype that covers the type of the values that can be specified by the formatting directives. The first element of the formatting item list replaced the leftmost formatting direcive, the second item on the list replaces the second leftmost formatting directive, and so on.

A formatting directive is introduced by the character %, after which the following may appear:

1. zero or more flags, modifying the meaning of the directive;
2. an optional minimal field width (decimal number), specifying a minimal amount of space to fill;
3. an optional precision, specifying the number of digits to the right of the decimal point for real numbres directed by *e*, *E* or *f* directives, or maximum number of significant digits for real numbers directed by *g* or *G* directives. The precision is specified by a decimal point (.) followed by a decimal number;
4. a single character specifying the type of directive (conversion).

To actually produce a % in the resulting string, you can use the special %% directive. Directives include:

d the formatting item *INT (n)*, *LINT (n)*, *WORD (n)*, *LWORD (n)* or *WORD8 (n)* is converted to a signed decimal;

x*,*X the formatting item *INT (n)* or *LINT (n)* is converted to signed hexadecimal; if *x* is specified, the letters *abcdef* are used in the hexadecimal representation, if *X* is specified, the letters *ABCDE-F* are used;

o the formatting item *INT (n)* or *LINT (n)* is converted to signed octal;

c the formatting item *CHR (c)* is converted to a single character string containing *c*;

b the formatting item *BOOL (b)* is converted to a string (*true* or *false*);

s the formatting item *STR (s)* inserts *s* in the output, the formatting item *ATOM (a)* inserts the string representation of the *a* in the output (via *Atom.toString*);

f the formatting item *REAL (r)* or *LREAL (r)* is converted to *[-]ddd.ddd*, with the number of digits to the right of the decimal point given by the precision (defaults to 6); a precision of 0 forces no printing of the decimal point (unless a # flag is specified, see below); if a decimal point is printed, at least one digit appears before it (possibly 0);

e*,*E as for *f*, except that the number is represented as *[-]d.ddde[-]dd* for *e*, and *[-]d.dddE[-]dd* for *E*;

g*,*G the formatting item *REAL (r)* or *LREAL (r)* is converted as if the directive was *f* or *e* (respectively *F* if *G* is used) .

The following flags may be used to change the behavior of a directive:

- the result of the conversion will be left-justified within the field specified by the minimal field width (default is to right-justify);

$\sim$ uses $\sim$ as a negation character (the default is to use -); thus, it is important to use the $\sim$ flag when printing out numbers meant to be read back by SML;

+ forces the result of numeric conversions to begin with a plus or minus sign;

(space) if the first character of a numeric conversion is not a sign, prefix a space to the result;

0 numeric conversions are zero-padded on the left, after any leading sign and base indicator (*x*,*0x*,*o*,*0o*,...); no effect on left-justified numeric conversions;

reduces the result in alternate form; the effect of this flag depends on the directive: for *o*, it forces the first digit to be zero; for *x* (or *X*), it forces a *0x* (or *0X*) to be prefixed (default is *x*); for *e*,*E*,*f*, a decimal point is always printed, even if no digits follow; for *g* or *G*, trailing zeros are not removed ; there is no effect on the other directives.

Note that it is an error to specify both a (space) and a + flag. In general, any error in format directives raises the *BadFormat* exception.

The values to be handled by the formatting directives are passed as a list of elements of type *fmt_item*, where the first element passed is associated with the leftmost format directive in the format string, and so on. Constructors for *fmt_item* map the underlying value by an indication of what it is, and can only correspond to some directives. An exception *BadFmtList* is raised if a mismatch occurs between formatting item and formatting directive. The format items *LINT*, *INT*, *LWORD*, *WORD* and *WORD8* are associated with *%d*, *%x*, *%X* and *%o*, *CHR* with *%c*, *BOOL* with *%b*, *ATOM* and *STR* with *%s*, and *REAL* and *LREAL* with *%e*,*%E*,*%f*,*%g* and *%G*. The specific formatting items *LEFT* and *RIGHT* take an integer *i* and a formatting item *f* and specify that the formatting item *f* should be left (respectively right) justified within a field of width *i* .

The function *formatf* in the structure *Format* is similar to *format*, but takes an extra argument. It takes a function *f* of type *string*$\rightarrow$*unit* which is called with the result of the formatting. The call *formatf s f l* is equivalent to *f (format s l)*. .

The *Format* structure provides the functionality to write formatted data into a string. The opposite direction, reading formatted data from a string, is provided by a structure *Scan*, matching the signature given in Figure 7.24.

```
structure 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)
      | RIGHT of (int * fmt_item)

  exception BadFormat

  val sscanf : string -> string -> fmt_item list option
  val scanf  : string -> (char, 'a) StringCvt.reader
                 -> (fmt_item list, 'a) StringCvt.reader

end
```

Figure 7.24: The structure *Scan*

```
structure ListFormat : sig

  val fmt : {
       init : string,
       sep : string,
       final : string,
       fmt : 'a -> string
     } -> 'a list -> string
  val listToString : ('a -> string) -> 'a list -> string
  val scan : {
       init : string,
       sep : string,
       final : string,
       scan : (char, 'b) StringCvt.reader -> ('a, 'b) StringCvt.reader
     } -> (char, 'b) StringCvt.reader -> ('a list, 'b) StringCvt.reader

end
```

Figure 7.25: The structure *ListFormat*

The *sscanf* function works similarly as *format* in *Format*, in that it takes a format string describing the format in which the data is expected. Directives just like those given for *format* can be specified in the format string, which will match the corresponding values in the input to be scanned .

The result of performing a scan is an option value: *NONE* if the input string to be scanned does not match the format string, or *SOME (l)* where *l* is a list of format items (of type *fmt_item list*) giving the values corresponding to the format directives in the scanned input string.

The function *sscanf* scans a string, but we saw in Section 4.3 that scanning a string can be generalized to scanning a stream of characters via a stream reader function. The function *scanf* generalizes *sscanf* in exactly that way. It takes a format string and a character stream reader (over any stream, of type *'a*), and returns a *fmt_item list* stream reader (over streams of type *'a*). The idea is simply that *sscanf* converts a stream of characters into a stream of list of values read from the stream according to the format string. As usual, *sscanf fmt* is equivalent to *StringCvt.scanString (scanf fmt)*.

A final piece of support for formatting concerns lists. The structure *ListFormat* (whose signature is given Figure 7.25) implements formatting and scanning functions for lists of elements. For formatting a list $[x_1,...,x_n]$ into a string, the idea is to provide strings specifying what goes at the beginning and the end of the list, the element separator to use, and a function to transform each element into a string. The function *fmt* takes all that information in the form of a record of type $\{$*init: string, final: string, sep:string, fmt:'a* $\rightarrow$*string*$\}$, expects an *'a list* $[x_1,...,x_n]$ and

```
structure RealFormat : sig

  val realFFormat : (real * int) -> {sign : bool, mantissa : string}
  val realEFormat : (real * int) -> {sign : bool, mantissa : string, exp : int}
  val realGFormat : (real * int) -> {sign : bool, whole : string, frac : string, exp : int option}

end
```

Figure 7.26: The structure *RealFormat*

returns essentially the string:

```
init^(fmt $x_1$)^sep^(fmt $x_2$)^sep^...^sep^(fmt $x_n$)^final
```

The function *listToString* formats a list into the style used by SML itself: the call *listToString fmt* is equivalent to *fmt* {*init="[",final="]",sep=",",fmt=fmt*}.

Scanning a list from a stream (given a reader for the stream, see Section 4.3) is done by the function *scan*, which takes as input a record of type {*init: string, final: string, sep: string, scan: (char,'b) StringCvt.reader* →*('a,'b) StringCvt.reader*} (with *init* and *final* the elements surrounding the list to be recognized, *sep* a string for the separator, and *scan* a function from a character reader on streams to *'a* readers on streams) and a character reader on streams and returns an *'a list* reader on streams. The idea is that the resulting list readers attempts to read the initial string, an element (using the *scan* function in the record), a separator, an element, and so on until the final string, at which point the appropriate list is returned (as well as the remainder of the stream, as usual). Whitespace is ignored during the scan. As with all stream readers, *NONE* is returned if an appropriate list cannot be scanned from the supplied stream.

For example, the following call creates an *int list* scanning function that scans SML-style integer lists:

```
ListFormat.scan {init="[",final="]",sep=",",scan=Int.scan}
```

As with other scanning functions, it can be used to create a reader by passing to it a character reader on a stream. The resulting reader returns integer lists read from the stream. The following call creates another *int list* scanning function, scanning Lisp-style integer lists:

```
ListFormat.scan {init="(",final=")",sep=" ",scan=Int.scan}
```

Finally, the structure *RealFormat* (signature given in Figure 7.26) provides low-level real to string conversion functions, which are used internally to implement functions such as *Real.fmt* and so on. The key is that instead of returning a string representing the whole real number, these functions return string representations of the key parts of the real representation (sign, mantissa, exponent, etc).

The function *realFFormat* takes a real number and a precision, and returns a record with a boolean value representing the sign of the real number, and a string representation of the digits of the real number. The precision indicates how many fractional digits to include (0's are appended if necessary)

The function *realEFormat* is similar, but instead of simply returning the sign and mantissa, it returns the exponent as well, when the real number is expressed in scientific notation. Note that in this case the mantissa is normalized . The precision as before indicates the number of fractional digits in the mantissa (0's are appended if necessary).

The function *realGFormat* returns the sign and string representations for the whole and fractional parts. An optional exponent is also returned, if the number needs to be expressed in scientific notation (i.e. if there are not enough digits in the real representation to actually represent the number). For this function, the precision represents the total number of significant digits in the whole and fractional parts. Trailing 0's in the fractional part are dropped.

.

7.8 Handling command-line arguments

When a program is executed from the operating system shell, it is often passed command-line arguments. For example, a text editor may be passed a file name to create or edit, as well as various command-line switches (or options) indicating that the program should do this and that upon startup. As we saw in Section **??**, when a heap is exported by *exportFn* and later loaded by SML/NJ, the exported function is passed a *string* $\times$*string list* argument containing the name under which it was called from the operating system, and a list of the command-line arguments used in the call. Such arguments may also be accessed by the Basis Library structure *CommandLine* (see Section 4.6).

At this point then, we know how command-line arguments are passed to the called program, and how to access them. The problem of how to handle them, how to process them, how to recognize them in a consistent way still remains. This general problem is exacerbated by the fact that users expect a certain behavior from command-line arguments, namely that it should be possible to specify both long and short names for the options, that some options should take arguments, that options may be specified in any order, or clumped together, or mixed freely with other command-line arguments such as filenames.

```
structure GetOpt : sig

  datatype 'a arg_order
        = RequireOrder
        | Permute
        | ReturnInOrder of string -> 'a
  datatype 'a arg_descr
        = NoArg of unit -> 'a
        | ReqArg of (string -> 'a) * string
        | OptArg of (string option -> 'a) * string
  type 'a opt_descr = {
          short : string,
          long : string list,
          desc : 'a arg_descr,
          help : string
        }

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

end
```

Figure 7.27: The structure *GetOpt*

Instead of reinventing the wheel every time a new program is written, developpers have come up with libraries of code to deal with command-line arguments in a uniform way. One such library for C, the so-called GNU GetOpt library, has rapidly become a standard. This library has been ported to SML and is available in the SML/NJ Library.

The structure *GetOpt* (whose signature is given in Figure 7.27) implements facilities for handling command-line arguments à la GNU GetOpt. The approach is to describe every command-line option of the program by a value of type *'a opt_descr*, which is a record of type {*short:string,long:string list,desc:'a arg_descr,help:string*}. The high-level idea is that the function to recognize command-line options is given a list of such *'a opt_descr* values and a command line, and returns essentially a list of *'a* values representing the options that were passed on the command line. The choice of *'a* is very much a design issue, but is typically a datatype. A command-line option can have multiple names, and invoked by either a short (one character) name introduced by a hyphen, like *-a*, or a long name introduced by a double hyphen, like --*add*. The field *short* is a string containing all the characters recognized as that option, i.e. *"ain"* means that the option can be specified as -*a*, -*i* or -*n* (typically, only one such character is given though). The field *long* contains a list of the strings for the recognized long names, i.e. *["add","inject","new"]* for --*add*, --*inject* and --*new*. The field *help* contains a description of the option which can be used by the system to give online help. The field *desc* describes the type of command-line option this particular option belong to. The possibilities are given by the *'a arg_descr* type.

An option with *desc* set to *NoArg (f)* specifies that this option does not take an argument. When such an option is recognized (we describe later how to recognize options), the function *f* (of type *unit* →*'a*) is invoked to return the value of type *'a* representing that option. An option with *desc* set to *ReqArg (f,argdesc)* specifies that this option takes a required argument, which must follow it on the command line. For options invoked with a short name, an argument looks like -*a somearg*, while for long names, arguments are passed as --*add*=*somearg*. When the option is recognized, the function *f* (of type *string*→*'a*) is called with the option argument. The string *argdesc* in the description *ReqArg* of the option is used for documentation purposes, and should be set to a short description (one word) of what the argument means (i.e. *"file"*, or *"name"*, or *"date"*). It is used by the system to generate online help. Finally, an option with *desc* set to *OptArg (f,argdesc)* specifies that this option takes an optional argument. When such an option is recognized, the function *f* (of type *string option* →*'a*) is called, and passed *NONE* if no argument was given on the command line (the option was followed by nothing or by another option for short names, or no =*val* for long names), or *SOME (s)* if an argument *s* was given. As with *ReqArg*, the *argdesc* value in *OptArg* is a string used by the system to generate online help, and should be set to a one-word description of the type of argument this option expects.

Two functions are provided to handle command-line options. The first function, *usageInfo*, synthesizes a helpful online help message, given a list of options that the program recognizes. This function should be called to generate and print a help message if for example an error is encountered while processing the command-line arguments. Formally, *usageInfo* takes a string header which is a piece of text reported verbatim (typically giving the syntax of the call, as in *"foo [options] filename"*) and a list of values describing the options, of type *'a opt_descr list*. For each such option, the system generates a line giving the names under which the option can be invoked, a description of the argument if one is needed, and the help text for that option.

As an example, consider the following list of option descriptions for an hypothetical application *edit*:

```
val options = [{short="h",
                long=["help"],
                desc=GetOpt.NoArg (fn () => HelpOption),
                help="produces this help message"},
               {short="nc",
                long=["new","create"],
                desc=GetOpt.NoArg (fn () => NewOption),
                help="create a new file, if one already exists"},
               {short="b",
                long=["background","bg"],
                desc=GetOpt.ReqArg (BgOption,"color"),
                help="background color of the editor"}]
```

We assume we have a datatype *option_result* returned by the option recognizer:

```
datatype option_result = HelpOption
                       | NewOption
                       | BgOption of string
```

Note that this means that the list *options* has type *option_result opt_descr list*. In this example, we shamelessly use the fact that a constructor *Foo of t* for a datatype *T* has type *t*→*T* (for use in *ReqArg* or *OptArg*).

Calling *usageInfo* with this list of options generates the following help message:

```
- GetOpt.usageInfo header="some header", options = options;
val it =
  "some header\n  -h        --help                          produces this #"
  : string
- print it;
some header
  -h        --help                          produces this help message
  -n, -c    --new, --create                 create a new file, if one already exists
  -b color  --background=color, --bg=color  background color of the editor          val it = () : uni
```

The function *getOpt* performs the actual conversion from command-line arguments to values of type *'a*, given an option description list of type *'a opt_descr list*. In the above example, *getOpt* would return essentially an *option_result list*. One way in which to understand this is that *getOpt* converts a list of strings representing command-line arguments into a list of values representing command-line arguments, values chosen to be easily interpretable by the program.

Actually, *getOpt* is more refined. It takes as input a record describing the operations to perform, of type {*options:'a opt_descr list, argOrder:'a ord_order, errorFn:string*→*unit*} where *options* is the list of options, as previously, *argOrder* is an element of the *'a arg_order* datatype, describing what to do with options following non-options, and can be one of:

RequiredOrder no option processing after a non-option; after the first non-option, options are treated as non-options;

Permute freely intersperse options and non-options;

ReturnInOrder (f) with f of type $string \rightarrow 'a$, where $'a$ is the type appearing in the type of *options*, *'a opt_descr list*; this flag indicates to *getOpt* that non-options should be wrapped using function f into options, and returned as such.

The full meaning of the names of these various flags will become clear when we look at what exactly *getOpt* returns. The last field in the first parameter to *getOpt* is *errorFn*, which defines a function to handle errors generated by *getOpt*. Common behaviors may include raising an exception, printing a warning and continuing to handle options, or printing the usage information (given by *usageInfo*).

The second argument to *getOpt*, after the record describing how to handle options, is the actual list of command-line arguments, as obtained for example by a call to *CommandLine.args*.[6] The result of *getOpt* is a pair of lists, the first list containing the values corresponding to the options (returned as an *'a list*, where *'a* is the type appearing in the type of the list of options, *'a opt_descr list*); the second list contains the unprocessed non-options (as a list of strings). What exactly is returned, and in what order, depends on the *argOrder* flag passed to *getOpt*. If *RequireOrder* was passed, *getOpt* returns all the options up to the first non-option in order in the first list, and everything else (including later options, unprocessed) in order in the second list. If *Permute* was passed, *getOpt* returns all the options specified on the command line in relative order in the first list (if one option came before another option on the command line, the former will appear before the latter in the resulting list), and returns all the non-options in relative order in the second list. Finally, if *ReturnInOrder* was passed, every option and every wrapped non-option is returned in order in the first list, and the second list is empty.

7.9 Miscellaneous functionality

We now examine the remaining modules available in the SML/NJ Library, which are not as neatly classifiable as the above ones. Not due to lack of applicability, far from that. But they go under the heading of general utility functions. They are presented in no particular order.

The structure *Iterate* (whose signature is given in Figure 7.28) implements simple higher-order functions to iterate over functions. The function *iterate* takes a function f of type $'a \rightarrow 'a$ and a count n, and creates the function $f^n = f \; o \; f \; o \; \ldots \; o \; f$. By definition, $f^0\,(a) = a$, $f^1\,(a) = f\,(a)$, $f^2\,(a) = f\,(f\,(a))$, and so on. The function *repeat* is similar, except that the function f passed as an argument has type $int \times 'a$

[6] The list of command-line arguments is also supplied to the function given during an *exportFn* (see Section 5.4).

```
structure Iterate : sig

  val iterate : ('a -> 'a) -> int -> 'a -> 'a
  val repeat : (int * 'a -> 'a) -> int -> 'a -> 'a
  val for : (int * 'a -> 'a) -> (int * int * int) -> 'a -> 'a

end
```

Figure 7.28: The structure *Iterate*

$\rightarrow$*'a* and the current "count" is passed to *f* every time it is called. In other words, if f_r^n = *repeat f n*, then f_r^0 *(a) = (0,a)*, f_r^1 *(a) = (1,f (0,a))*, f_r^2 *(a) = (2,f (1,f (0,a)))*, and so on. The intuition is that *f* can act differently at every iteration. This is meant to model (some kinds of) repeat-loops in imperative languages where, intuitively, the function *f* passed as an argument represents the body of the repeat-loop, and the first argument to *f* represent the variable over which the repetition is performed, accessible at every iteration of the loop. This of course begs the question: could we model general repeat-loops in such a way? A general repeat loop repeats until a given condition is satisfied. If we assume that the condition can involve the current iteration count and the value of the body at that iteration, we can implement a *repeatUntil* as:

```
fun repeatUntil (f:int * 'a -> 'a) (until:int * 'a -> bool) (a:'a) = let
  fun loop (i,curr) = if (until (i,curr))
                        then curr
                      else loop (i+1,f (i+1,curr))
in
  loop (0,f (0,a))
end
```

This description of *repeat* is very much reminiscent of a fold. In fact, it is easy to see that *repeat f n* is equivalent to *foldl f 0 [1,2,...,n-1]*, except that we do not explicitly construct the list *[1,2,...,n-1]*.

The last function in *Iterate* is *for*, which models the behavior of for-loops in imperative languages, at least when the induction variable is an integer. The function *for* performs just like a *repeat*, but while *repeat f n* loops with indices 0 up to $n-1$, *for f* (n_1,n_2,n_3) loops with indices $n_1, n_1+n_3, n_1+2n_3, \ldots, n_1+kn_3$ (where $k = \lfloor \frac{n_2-n_1}{n_3} \rfloor$). (If $n_3 < 0$, the behavior is similar, but in the opposite direction.) As in the case of *repeat*, the function *f* passed to *for* is passed the current count at every iteration. Note that *for* can also be viewed as a fold, but this time over the list $[n_1+n_3,...,n_1+kn_3]$, with base value n_1.

For all the functions in *Iterate*, the exception *LibBase.Failure* is raised if the loop limits are not correct. For *iterate* and *repeat*, this happens if $n < 0$; for *for*, if $n_3 \geq 0$ and $n_1 > n_2$, or if $n_3 \leq 0$ and $n_1 < n_2$.

```
structure ListXProd : sig

  val appX : (('a * 'b) -> 'c) -> ('a list * 'b list) -> unit
  val mapX : (('a * 'b) -> 'c) -> ('a list * 'b list) -> 'c list
  val foldX : (('a * 'b * 'c) -> 'c) -> ('a list * 'b list) -> 'c -> 'c

end
```

Figure 7.29: The structure *ListXProd*

Just as we did for *repeat*, can we come up with a generic for-loop, that allows looping over values different than integers? We can, at the cost of not being able to statically determine non-termination of the loop (using integers allows us to catch non-terminating loops). Consider:

```
fun for (f:'a * 'b -> 'b) {start:'a, term:'a -> bool, inc:'a -> 'a} (b:'b) = let
  fun loop (a:'a, curr:'b):'b =
    if term (a)
      then curr
    else let
      val next = inc (a)
    in
      loop (next,f (next,curr))
    end
in
  loop (start,b)
end
```

The structure *ListXProd* (signature in Figure 7.29) provides the simple iterator functions *appX*, *mapX* and *foldX* over the cross-produce of two lists. The cross-product of two lists *[*x_1*,...,*x_n*]* and *[*y_1*,...,*y_m*]* is here taken to mean the list *[(*x_1*,*y_1*), ..., (*x_1*,*y_m*), (*x_2*,*y_1*), ..., (*x_2*,*y_m*), ..., (*x_n*,*y_1*), ..., (*x_n*,*y_m*)]*. The function *appX* simply applies a function to the elements of the cross-product, while *mapX* creates a new list after mapping a function to each element of the cross-product. (The call *mapX f (x,y)* produces a list of size *length (x) * length (y)*.) The function *foldX* folds a function *f* on the left over the cross-product of the lists. Note that the behavior of cross-product functions are quite different than those in structure *ListPair*, which consider the elements of two lists in order, two at a time: the elements at position 0 in the two lists, the elements at position 1 in the two lists, etc.

The structure *IOUtil* (signature in Figure 7.30) defines useful functions to automatically perform input and output redirection. The function *withInputFile* takes a filename *s*, a function *f* and an argument *a* to which to apply the function *f*, and evaluates *f (a)* under a context where *TextIO.stdIn* refers to the input stream created from file *s*. After *f (a)* is evaluated, *TextIO.stdIn* is reset to what it was prior to the call. The function *withOutputFile* does a similar thing, but rebinds *TextIO.stdOut*

```
structure IOUtil : sig

  type instream
  type outstream

  val withInputFile : string * ('a -> 'b) -> 'a -> 'b
  val withInstream : instream * ('a -> 'b) -> 'a -> 'b
  val withOutputFile : string * ('a -> 'b) -> 'a -> 'b
  val withOutstream : outstream * ('a -> 'b) -> 'a -> 'b

end
```

Figure 7.30: The structure *IOUtil*

```
structure PathUtil : sig

  val findFile  : string list -> string -> string option
  val findFiles : string list -> string -> string list
  val existsFile : (string -> bool) -> string list -> string -> string option
  val allFiles   : (string -> bool) -> string list -> string -> string list

end
```

Figure 7.31: The structure *PathUtil*

instead. The functions *withInstream* and *withOutstream* rebind *TextIO.stdIn* and *TextIO.stdOut* to the supplied input and output streams respectively. Thus, any function that takes its input from *TextIO.stdIn* can be made to take its input from an arbitrary file or stream, and similar any function that prints to *TextIO.stdOut* (say, via the function *TextIO.print*) can be made to print its output to an arbitrary file or stream. Note that the types *instream* and *outstream* are transparently bound to their *TextIO* counterparts.

The structure *PathUtil* (signature in Figure 7.31) provides some simple higher-level pathname and searching utilities. The function *existFile* takes a predicate on filenames p, a list of pathnames $p_1,...,p_n$ and a filename f, and applies to p to p_1/f up to p_n/f, until one such test evaluates to *true*. If no test succeeds, *NONE* is returned, otherwise *SOME* (p_i/f) for the test that succeeded. The function *allFiles* is similar, except that it returns all the filenames satisfying the predicate. As a simple example of the use of these functions, the function *findFile* and *findFiles* are provided that simply check whether the given file exists in any of the directories pointed to by the paths, or respectively returns all such files that exist in those directories. Those two functions are easily written in terms of *existFile* and *allFiles*.

```
structure Random : sig

  type rand

  val rand : (int * int) -> rand
  val toString : rand -> string
  val fromString : string -> rand
  val randInt : rand -> int
  val randNat : rand -> int
  val randReal : rand -> real
  val randRange : (int * int) -> rand -> int

end
```

Figure 7.32: The structure *Random*

The structure *Random* (signature in Figure 7.32) implements a random number generator (really, a pseudo-random number generator). The type *rand* represents the internal state of the random number generator, state which is used to compute the next random number. The state is initially "seeded" by a call to the function *rand* with two integers arguments. Random numbers are generated deterministically from the internal state, and therefore internal states seeded with the same values will generate the same sequence of random numbers, which can be useful for debugging. On the other hand, to non-deterministically generate numbres, it may be useful to seed the state with unpredictable values, such as can be derived from the time of day. For example:

```
rand (Int32.toInt (Time.toSeconds (Time.now ())),0)
```

The functions *fromString* and *toString* respectively convert an internal state from and to a string. This is useful to save the current state to a file, and restoring the state later.

Four functions are provided to generate random numbers. All take a state as an argument, and all modify the passed state as a side effect. The function *randInt* returns a random integer between *Int.minInt* and *Int.maxInt*. The function *randNat* generates a random natural number between *0* and *Int.maxInt*. The function *randRange* applied to *(lo,hi)* generates a random integer between *lo* and *hi*. Finally, *randReal* generates a random real number between *0.0* and *1.0*.

While most structures in the SML/NJ Library are rather independent of SML/NJ itself, the structure *TimeLimit* (signature in Figure 7.33) relies on the timing facilities of SML/NJ and the support for signals (see Chapter **??**). The structure *TimeLimit* provides a function to do timeouts on function application. The call

```
structure TimeLimit : sig

  exception TimeOut

  val timeLimit : Time.time -> ('a -> 'b) -> 'a -> 'b

end
```

Figure 7.33: The structure *TimeLimit*

timeLimit t f x (with *t* a *Time.time* value) will evaluate *f (x)* but timeout after elasped time *t* if the evaluation is not finished by then. This can be very useful to prevent non-essential and potentially long-running computations from taking too much time. If such timing out is intrinsic to the application, it may be an indication that what you need is concurrency facilities, and you should then probably turn to CML, described in an future version of these notes.

The structure *ParserComb* (signature in Figure 7.34) provides straightforward constructs for building simple parsers. We will return to parsing in more depth in Chapter **??** when we introduce ML-Yacc, but for now let us just say that parsing consists of determining whether a given sequence of tokens is part of the language of a given grammar. The type of a parser according to *ParserComb* is *('a,'strm) parser*, which is just an abbreviation for the type *(char,'strm) StringCvt.reader* →*('a,'strm) StringCvt.reader*. In other words, an *('a,'strm) parser* is just a scanning function for *('a,'strm)* values. The *ParserComb* structure provides combinators to build complex parsers from simple ones.

The simplest parser is *result (v)*, which is a parser that always returns the value *v* (of type *'a*), without touching the stream. Similarly, the parser *failure* always returns an indication that the scanning failed (the value *NONE*). Slightly more involved are the parsers created by *char (c)* and *string (s)*, which respectively scan a given character from the stream, or a given string from the stream. The function *eatChar (p)* takes a predicate on characters and returns a parser that scans the next character from the stream if it satisfies the predicate, and fails otherwise. More generally, the function *token (p)* takes a predicate on characters and returns a string parser, that scans the longest string containing all the characters satisfying the predicate.

The above functions all create simple parsers from scratch. The remaining functions combine parsers to form more complex ones. The combinator *wrap* takes an *('a,'strm) parser* and a transformer function of type *'a*→*'b*, and returns a *('b,'strm) parser* that uses the *('a,'strm) parser* to get *'a* values from the stream and transforms them. The *seq* combinator takes two parsers for values of type *'a*

```
structure ParserComb : sig

  type ('a, 'strm) parser =
    (char, 'strm) StringCvt.reader -> ('a, 'strm) StringCvt.reader

  val result : 'a -> ('a, 'strm) parser
  val failure : ('a, 'strm) parser
  val wrap : (('a, 'strm) parser * ('a -> 'b)) -> ('b, 'strm) parser
  val seq : (('a, 'strm) parser * ('b, 'strm) parser) -> (('a * 'b), 'strm) parser
  val seqWith : (('a * 'b) -> 'c)
                  -> (('a, 'strm) parser * ('b, 'strm) parser)
                    -> ('c, 'strm) parser
  val bind : (('a, 'strm) parser * ('a -> ('b, 'strm) parser))
               -> ('b, 'strm) parser
  val eatChar : (char -> bool) -> (char, 'strm) parser
  val char   : char -> (char, 'strm) parser
  val string : string -> (string, 'strm) parser
  val skipBefore : (char -> bool) -> ('a, 'strm) parser -> ('a, 'strm) parser
  val or : (('a, 'strm) parser * ('a, 'strm) parser) -> ('a, 'strm) parser
  val or' : ('a, 'strm) parser list -> ('a, 'strm) parser
  val zeroOrMore : ('a, 'strm) parser -> ('a list, 'strm) parser
  val oneOrMore  : ('a, 'strm) parser -> ('a list, 'strm) parser
  val option : ('a, 'strm) parser -> ('a option, 'strm) parser
  val token : (char -> bool) -> (string, 'strm) parser

end
```

Figure 7.34: The structure *ParserComb*

and*'b*, and returns a new parser that attempts to read a value of type *'a* followed by a value of type *'b* using the two parsers in sequence. The result is a pair of values of type *'a* $\times$ *'b*. If such a sequence of values cannot be found, the parser fails. More generally, *seqWith* is like *seq*, except that it also takes a transformer function *'a tuple 'b* $\rightarrow$ *'c* which is applied to the two parsed value to get a value of type *'c*. The combinator *bind* implements dependent parsing. It takes a *('a,'strm) parser* and a function *'a* $\rightarrow$*('b,'strm) parser*, and creates a *('b,'strm) parser* that behaves as follows: it tries to parse a value of type *'a* from the stream, and uses that value to create a *('b,'strm) parser* to parse a value of type *'b* from the stream.

To conclude our description of the provided functions, the combinator *skipBefore* takes a character predicate *p* and an *('a,'strm) parser* and returns a new *('a,'strm) parser* that skips any character satisfying the predicate before trying to parse a value of type *'a* using the supplied parser. The combinator *or* takes two *('a,'strm) parser* and creates a parser that successively attempts to parse a value of type *'a* using one of the parsers. Whichever succeeds, its value is returned. Only one value is returned. The *or'* combinator is a generalization of *or* to list of parsers, and works under the same principle. Note that *or* (and *or'*) are not backtracking choices. Once a branch of the *or* that has been selected and has successfully parsed a value, a subsequent failure of the parse will not backtrack to the *or* to try the other branch. Similarly to *or*, *oneOrMore* (respectively *zeroOrMore*) takes a parser and attempts to parse at least one (respectively zero) value from the stream using the parser, until the parsing fails. The result is returned as a list of the parsed values. Finally, *option* turns a parser into a parser that never fails: it simply injects *NONE* in the resulting stream when the parser would fail.

Notes

A good reference for algorithmic and data structure issues is the brick by Cormen, L and Rivest [**?**].

Interesting issues regarding the typing of format strings (such as those found in *Format*) are described by Danvy [**?**].

Atoms are related to symbols as found in Lisp/Scheme.

The implementation of red-black trees used by the SML/NJ Library is fully functional, and is described by Okasaki in [**?**]. More details about the implementation of data structures using purely functional algorithms can be found in Okasaki's book [**?**].

Splay trees are described in [**?**].

Property lists have been introduced by Stephen Weeks and used in the MLton compiler to keep information between compiler passes. The implementation uses a little-known subtletly in the SML type system: the fact that the exception type *exn* can be used as an extensible datatype.

Sorting algorithms are described in most introductory computer science textbooks.

The implementation of MergeSort used in the SML/NJ Library is from Paulson [**?**]. The implementation of QuickSort is a highly engineered version due to Bentley and McIlroy [**?**].

Details on hashing functions and such.

GetOpt reference from GNU.

Source of *Random* structure. Another structure for pseudo-random number generation is provided, *Rand*, but is not as good as *Random*. It is based on [**?**].

Parser combinators have been described in detail by Hutton and Meijer [**?**]. They form a monad [**?**].

Bibliography

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

[2] R. Adams, W. Tichy, and A. Weinert. The cost of selective recompilation and environment processing. *ACM TOSEM*, 3(1), 1994.

[3] A. Appel, D. MacQueen, R. Milner, and M. Tofte. Unifying exceptions with constructors in standard ML. Technical Report ECS LFCS 88 55, Laboratory for Foundations of Computer Science, Department of Computer Science, Edinburgh University, June 1988.

[4] A. W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3:343–380, 1990.

[5] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[6] A. W. Appel. Axiomatic bootstrapping: A guide for compiler hackers. *ACM Transactions on Programming Languages and Systems*, 16(6):1699–1718, 1994.

[7] A. W. Appel and D. B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 301–324. Springer-Verlag, September 1987.

[8] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Third International Symposium on Programming Languages Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, August 1991.

[9] A. W. Appel and D. B. MacQueen. Separate compilation for Standard ML. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23. ACM Press, 1994.

[10] L. Augustsson. A compiler for Lazy ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227. ACM Press, 1984.

[11] L. Augustsson and T. Johnsson. Lazy ML user's manual. Technical report, Department of Computer Science, Chalmers University of Technology, 1987.

[12] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Studies in Logic. North-Holland, Amsterdam, 1981.

[13] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[14] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner. The ML Kit (version 1). DIKU-report 93/14, Department of Computer Science, University of Copenhagen, 1993.

[15] M. Blume. *Hierarchical Modularity and Intermodule Optimization*. PhD thesis, Princeton University, 1997.

[16] M. Blume. Dependency analysis for Standard ML. *ACM Transactions on Programming Languages and Systems*, 21(4), 1999.

[17] M. Blume and A. W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4), 1999.

[18] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Computer Science, University of Utah, 1992.

[19] R. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058, 1977.

[20] R. Burstall, D. MacQueen, and D. Sannella. HOPE: An experimental applicative language. In *Conference Record of the 1980 Lisp Conference*, pages 136–143, 1980.

[21] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of System F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994.

[22] A. Church. *The Calculi of Lambda Conversion*. Number 6 in Annals of Mathematical Studies. Princeton University Press, Princeton, NJ, 1941.

[23] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics in the NuPRL Proof Development System*. Prentice-Hall, 1986.

[24] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8, May 1987.

[25] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–63. ACM Press, 1999.

[26] N. Dershowitz and E. M. Reingold. *Calendrical Calculations*. Cambridge University Press, 1997.

[27] D. Duggan and C. Sourelis. Parametrized modules, recursive modules and mixin modules. In *1998 ACM SIGPLAN Workshop on ML*, 1998.

[28] S. I. Feldman. *Make – a Program for Maintaining Computer Programs*. Bell Laboratories, 1979.

[29] M. Felleisen and D. P. Friedman. *The Little MLer*. The MIT Press, 1998.

[30] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 153–162. ACM Press, 1998.

[31] K. Fisher and J. H. Reppy. The design of a class mechanism for MOBY. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1999.

[32] M. J. Fisher. Lambda-calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, pages 104–109, 1972.

[33] M. J. Fisher. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):257–286, 1993.

[34] M. Flatt. *Programming Languages for Reusable Software Components*. PhD thesis, Rice University, 1999.

[35] E. Gansner and J. H. Reppy, editors. *The Standard ML Basis Library*. Cambridge University Press, 2000. In preparation.

[36] L. George, F. Guillame, and J. H. Reppy. A portable and optimizing back end for the SML/NJ compiler. 1994.

[37] S. Gilmore. Programming in Standard ML '97: A tutorial introduction. Technical report ECS-LFCS-97-364, LFCS, Department of Computer Science, University of Edinburgh, 1997. Available from `http://www.dcs.ed.ac.uk/home/stg/NOTES/`.

[38] S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without dynamic types. Technical report ECS-LFCS-97-378, LFCS, Department of Computer Science, University of Edinburgh, 1997.

[39] J-Y. Girard. *Interprétation Fonctionelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, 1972.

[40] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[41] M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[42] The ML2000 Working Group. Principles and a preliminary design for ML2000. Unpublished manuscript, 1999.

[43] M. R. Hansen and H. Rischel. *Introduction to Programming using SML*. Addison Wesley, 1999.

[44] R. Harper. Programming in Standard ML. Online tutorial notes available from `http://www.cs.cmu.edu/~rwh/introsml/index.html`, 1998.

[45] R. Harper, P. Lee, F. Pfenning, and E. Rollins. A compilation manager for Standard ML of New Jersey. In *1994 ACM SIGPLAN Workshop on ML and its Applications*, pages 136–147, 1994.

[46] R. Harper, P. Lee, F. Pfenning, and E. Rollins. Incremental recompilation for Standard ML of New Jersey. Technical Report CMU-CS-94-116, Departement of Computer Science, Carnegie Mellon University, 1994.

[47] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules and sharing. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1994.

[48] R. Harper and J. C. Mitchell. The type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):11–252, 1993.

[49] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the AMS*, 146:29–60, 1969.

[50] L. Huelsbergen. A portable C interface for Standard ML of New Jersey. Technical memorandum, AT&T Bell Laboratories, November 1995.

[51] J. Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 53–96. Springer-Verlag, 1995.

[52] R. J. M. Hughes. Lazy memo functions. In *Proceedings, IFIP Conference on Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 129–146, 1985.

[53] IEEE. IEEE standard for binary floating-point arithmetic. Std 754-1985 (Reaffirmed 1990), 1985.

[54] IEEE/ANSI. IEEE standard for radix-independent floating-point arithmetic. Std 854-1987, 1987.

[55] W. Kahan. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. Work in progress. Available from `http://www.cs.berkeley.edu/~wkahan/ieee754status/`, 1996.

[56] R. Kelsey, W. Clinger, and J. Rees (Eds). The revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[57] A. R. Koenig. An anecdote about ml type inference. In *Proceedings of the USENIX Symposium on Very High Level Languages*. USENIX, 1994.

[58] X. Leroy. Unboxed objects and polymorphic typing. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 177–188. ACM Press, 1992.

[59] X. Leroy. Polymorphism by name. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1993.

[60] X. Leroy. Manifest types, modules, and separate compilation. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 109–122. ACM Press, 1994.

[61] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

[62] X. Leroy. Camlidl user's manual. Online manual available from `http://caml.inria.fr/camlidl/`, 1999.

[63] X. Leroy, J. Vouillon, and D. Doligez. The Objective Caml system. Software and documentation available from `http://pauillac.inria.fr/ocaml/`, 1996.

[64] D. MacQueen. Modules for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–207. ACM Press, 1984.

[65] D. MacQueen. Using dependent types to express modular structure. In *Conference Record of the Thirteenth Annual ACM Sumposium on Principles of Programming Languages*, 1986.

[66] D. B. MacQueen. Structures and parameterisation in a typed functional language. In *Functional Programming Languages and Computer Architecture*, 1981.

[67] D. B. MacQueen and M. Tofte. A semantics for higher-order functors. In *Fifth European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1994.

[68] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[69] J. McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962.

[70] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In *Proceedings FPCA'91*, volume 523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[71] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17(3):348–375, 1978.

[72] R. Milner. A proposal for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 184–197. ACM Press, 1984.

[73] R. Milner and M. Tofte. *Commentary on Standard ML*. The MIT Press, Cambridge, Mass., 1991.

[74] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass., 1990.

[75] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Mass., 1997.

[76] J. C. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 270–278. ACM Press, 1991.

[77] E. Morcos-Chounet and A. Conchon. PPML, a general formalism to specify prettyprinting. In *Proceedings of the IFIP Congress*, Dublin, 1986. North-Holland.

[78] J. H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, Cambridge, Mass., 1968.

[79] J. H. Morris. A bonus from van Wijngaarden's device. *Communications of the ACM*, 15(8):773, 1972.

[80] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. Talx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for Systems Software (WCSSS '99)*, pages 25–35, 1999.

[81] G. Morrisett and R. Harper. Semantics of memory management for polymorphic languages. In *Higher-Order Operational Techniques in Semantics*, pages 175–226. Cambridge University Press, 1997.

[82] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 3(21):528–569, May 1999.

[83] G. Nelson, editor. *System Programming with Modula-3*. Prentice Hall Series in Innovative Technology. Prentice-Hall, 1991.

[84] Oberon Microsystems Inc. Component Pascal language report. Available from `http://www.oberon.ch`, 1997.

[85] D. C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[86] L. C. Paulson. *Isabelle, A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[87] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996.

[88] J. Peterson and K. Hammond (Eds). Report on the programming language Haskell, version 1.4. Technical report, Department of Computer Science, Yale University, 1997. Available from `http://www.haskell.org`.

[89] R. Pucella. The design of a COM-oriented module system. In *Proceedings of the Joint Modular Languages Conference*, number 1897 in Lecture Notes in Computer Science, pages 104–118. Springer-Verlag, 2000.

[90] R. Pucella and J. H. Reppy. An abstract IDL mapping for Standard ML. In preparation, 2000.

[91] C. Reade. *Elements of Functional Programming*. Addison Wesley, Reading, MA, 1989.

[92] D. Rémy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Conference Record of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 40–53. ACM Press, 1997.

[93] J. H. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, 1993.

[94] J. C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation*, number 19 in Lecture Notes in Computer Science, pages 408–425. Springer-Verlag, 1974.

[95] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1), 1965.

[96] C. V. Russo. *Types for Modules*. PhD thesis, University of Edinburgh, 1998. Available as LFCS thesis ECF-LFCS-98-389.

[97] C. V. Russo. First-class structures for Standard ML. In *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 336–350. Springer-Verlag, 2000.

[98] Z. Shao. Typed common intermediate format. In *Proceedings of the 1997 USENIX Conference on Domain Specific Languages*, pages 89–102. USENIX, 1997.

[99] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–129. ACM Press, 1995.

[100] S. Sokolowski. *Applicative High Order Programming: The Standard ML Perspective*. Chapman & Hall Computing, London, 1991.

[101] G. L. Steele Jr. RABBIT: A compiler for Scheme. AI Memo 474, MIT, 1978.

[102] G. L. Steele Jr. *Common Lisp the Language*. Digital Press, second edition, 1990.

[103] C. Szyperski. Import is not inheritance — why we need both: Modules and classes. In *European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 19–32. Springer-Verlag, 1992.

[104] C. Szyperski. *Component Software*. Addison Wesley, 1997.

[105] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192. ACM Press, 1996.

[106] W. F. Tichy. RCS – a system for version control. *Software — Practice and Experience*, 15:637–654, 1985.

[107] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1), 1990.

[108] M. Tofte and J-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[109] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[110] J. D. Ullman. *Elements of ML Programming*. Prentice-Hall, ML97 edition, 1998.

[111] P. Wadler. List comprehensions. In S. L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[112] P. Wadler. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1989.

[113] P. Wadler. A prettier printer. Unpublished manuscript, 1998.

[114] P. R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science. Springer-Verlag, 1992.

[115] N. Wirth. *Programming in MODULA-2*. Springer-Verlag, second edition, 1982.

[116] N. Wirth. From Modula to Oberon. *Software — Practice and Experience*, 18(7), 1988.

[117] A. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

[118] X/Open Company Ltd. X/Open Preliminary Specification X/Open DCE: Remote Procedure Call. 1993.

Appendix A

SML/NJ Grammar