The ML Kit
Version 1

Lars Birkedal Nick Rothwell Mads Tofte David N. Turner

March 17, 1993

Contents

Preface

The ML Kit is an implementation of the programming language Standard ML[11], written in Standard ML. In this preface we wish to explain why we have written the Kit, and for whom.

Historical origins

The ML Kit has its roots in the work on the formal definition of Standard ML, which is documented in the Definition of Standard ML [11] and the Commentary on Standard ML [10]. Writing the language definition involved "clerical" (although important) tasks, such as imposing structure and modularity on the semantics, choosing notation and terminology, and doing quite a lot of engineering with inference rules. These activities resemble programming. The thought occurred, that it might be feasible to spell out the entire language definition, right down to the choice of variable names, in Standard ML itself. This is precisely what we have undertaken to do in the ML Kit.

Relationship to other ML implementations

There are several very fine implementations of Standard ML, including Standard ML of New Jersey[3], Poly/ML and Poplog. These compilers are production quality compilers that generate efficient, optimized machine code, using all the tricks of the trade of compiler writing. By contrast, the ML Kit is basically an interpreted system, even though it does include a compiler into a simple call-by-value lambda-calculus. Therefore, if you are primarily looking for an efficient way of executing your Standard ML programs, the ML Kit is not the implementation for you. Having said that, the ML Kit is more efficient that one would think, given that it is written without any attempt to make it run fast.

The style of Kit programming

The style of Kit programming is extreme in several respects. First of all, Standard ML modules are used extensively, pressing the modules system to its limits. Indeed, there is a great deal more modularity in the Kit than in the language definition, because we want people to be able to pick and choose from the modules in the Kit, when they want to write programs that manipulate ML programs. Consequently, every module is a closed functor, i.e. it contains no free identifiers, except those that are part of the initial basis of Standard ML or are in the Edinburgh SML Library [7]. References and assignments are generally avoided, except for trivial purposes, such as generating fresh symbols. Whenever faced with the choice between writing clever code and writing clear code, we have consistently chosen not to write clever code (whether the code is clear, is not for us to say). No attempt is made to choose fast data structures and algorithms.

However, considerable energy has been devoted to achieving good modularity. For example, although many algorithms in the Kit use association lists for keeping information about identifiers (rather than hash tables or search trees), none of the algorithms knows it; they simply know that there are environments and that one can use certain operations on environments, but they do not know about the actual implementation details. That we can make such categorical statements about who knows what, is thanks to the modules system, in particular the functor concept, which makes it possible to get the ML system to check that modules do not use information which is not explicitly specified in their interfaces. You will probably never have seen as many sharing specifications in your life as you will see in the Kit, but that is the price one pays for having complete interfaces.

One of the main aims of writing the Kit has been to create a very "open" system, which people will think of as a collection of relatively independent modules, with clean interfaces, rather than a big, monolithic compiler. Indeed that is why the Kit is called the Kit.

Who is the Kit aimed at?

We imagine that a typical user of the Kit is a person who needs a parser and type-checker for ML, but has no intention of writing them. Another typical user would be somebody who would like to implement some variant of ML typing, for example polymorphism with a different way of type-checking records. Yet another possible user would be a person who wants to investigate the relative frequency of certain language constructs in real ML programs, and therefore needs a parser which produces abstract syntax trees that conform to the language definition. Finally, anybody who wants to write a compiler from ML to some other language, would of course be a potential user.

How to read this document

We assume that the reader has a general knowledge of Standard ML. Also necessary are a copy of the Definition of Standard ML and the Commentary on Standard ML (which we shall refer to simply as the Definition and the Commentary, from now on). We have not undertaken explaining the semantics of ML in this document; instead we frequently refer to parts of the Commentary when discussing details of the Kit.

The purpose of this document is not to give a thorough description of all the modules of the Kit. On the contrary, the idea is to supply just enough information that it becomes possible to navigate around the Kit.

We have tried to keep this document fairly modular. We recommend starting with Section 1, which is a quick tutorial to give an idea of how one uses the Kit to execute ML programs. Then proceed to Section 2 which explains the style of programming that connects the Kit and the Definition; this section also explains the overall structure, gives examples of Kit modules and explains how the whole Kit is put together.

Thereafter, one can branch out to the other parts of the document, depending on interest. We cover parsing, elaboration, evaluation, and compilation into the lambda-language.

Section A.3 explains how to install the Kit. Appendix B contains a list of things that are not (yet) implemented in the Kit, although they are Standard ML. Appendix B contains a list of known bugs.

What you must not expect of the Kit

The Kit was written mostly for fun; although it has coped successfully with large ML programs, is has never been rigorously tested, let alone proved to be correct with respect to the Definition. Even a careful testing of the Kit has been out of the question, given the limited manpower available for this project. The fact that the Kit is derived from the Definition is of course in itself no guarantee of its correctness.

Chapter 1

A first session with the Kit

Before diving into the internal structure of the Kit, let us explain how one uses the Kit for executing ML programs and how one performs simple changes to the Kit. We do so in the style of a tutorial; it is recommended that the reader try out the examples in this section on the machine. In the examples, we assume that SML of New Jersey is used, and that both the batch implementation and the interpreter has been built (see Section A.3). If Poly/ML is used to build the Kit, it is not possible to make the batch version and you should skip section 1.2 below.

1.1 Where to find things

Once the Kit has been built (see Section A.3), the directory `src` contains (at least) two executable files, `evalFile` and `kit`. The former is a stand-alone batch implementation of ML; the latter is the version one uses when one wants to modify the Kit. In the following subsections, we illustrate the use of both.

1.2 Running the Kit as a stand-alone program

You can use the Kit as a batch system which reads its input from a named file and prints the output on the standard output.

Example 1.1 The directory `examples` contains a the file `simpleprog.sml`, which contains the ML program

```
structure ARITH =
   struct
      datatype NAT = Zero | Succ of NAT
      fun twice(Zero) = Zero
        | twice(Succ x) = Succ(Succ(twice x))
   end ;

open ARITH ;
```

```
val two = Succ(Succ Zero) ;

twice two ;
```

which incidentally is the program which is studied in depth in Section 1 of the Commentary. Place yourself in the `src` directory; then type

```
          evalFile ../examples/simpleprog.sml
```

This executes the program in `simpleprog.sml` and prints the result on the terminal:

```
> structure ARITH =
    struct
      datatype NAT
        con Succ : (NAT -> NAT)
        con Zero : NAT
      val twice = fn : (NAT -> NAT)
    end
> datatype NAT
    con Succ : (NAT -> NAT)
    con Zero : NAT
  val twice = fn : (NAT -> NAT)
> val two = Succ(Succ(Zero)) : NAT
> val it = Succ(Succ(Succ(Succ(Zero)))) : NAT
```

1.3 Working with the Kit inside another ML system

When you want to modify the Kit, you work inside another ML system, for instance SML of New Jersey or Poly/ML. In the present example, we assume that SML of New Jersey is used. When you run the executable file `kit`, you then get a session with Standard ML of New Jersey (SML/NJ). This session, which we refer to as the *kit session*, differs from an ordinary SML/NJ session in that it has the Edinburgh Standard ML Library and the ML Kit modules loaded. In a kit session you can re-compile parts of the Kit using the ML `make` system (which is part of the Edinburgh Standard ML Library), you can execute the Kit on programs and you can create stand-alone programs, like the `evalFile` program shown in Section 1.1. We now illustrate each of these.

1.3.1 Starting `kit`

Place yourself in the directory `src` and type

```
          kit
```

which yields the result

```
The ML Kit, Version 1
Copyright (C) 1993 Edinburgh and Copenhagen Universities
val it = () : unit
-
```

which is the kit session's way of greeting you. If you now type

```
evalFile "../examples/simpleprog.sml";
```

you get the reply

```
> structure ARITH =
    struct
      datatype NAT
        con Succ : (NAT -> NAT)
        con Zero : NAT
      val twice = fn : (NAT -> NAT)
    end
> datatype NAT
    con Succ : (NAT -> NAT)
    con Zero : NAT
  val twice = fn : (NAT -> NAT)
> val two = Succ(Succ(Zero)) : NAT
> val it = Succ(Succ(Succ(Succ(Zero)))) : NAT
```

Notice that `evalFile` is now an ML function of type `string -> unit`. In total, the Kit only provides 6 functions:

```
sig
  val parse: unit -> unit
  val elab: unit -> unit
  val eval: unit -> unit

  val parseFile: string -> unit
  val elabFile: string -> unit
  val evalFile: string -> unit
end
```

The first three functions start up a top-level interactive dialogue with you, similar to an ML session with any other ML system. `parse` only parses, `elab` parses and elaborates, and `eval` parses, elaborates and evaluates. The dialogue is terminated by `^D`, which sends you back to the SML/NJ session.

```
- eval();
. val x = 3+3;
> val x = 6 : int
. val y = x + x;
```

```
> val y = 12 : int
. val it = () : unit
-
```

The first and the last line are part of the NJ/SML session; `^D` was typed after `.` in the penultimate line.

The last three functions, `parseFile`, `elabFile` and `evalFile`, correspond to `parse`, `elab` and `eval`, except that the former take their input from a file, whose name is provided as an argument.

1.3.2 Modifying the Kit

When you want to modify the Kit, the basic mode of operation is as follows. You first edit the source file(s) you want to change, using your favourite editor. You then tell the kit session that you have touched these files and ask it to re-compile as little as possible in order to re-build the Kit.

Example 1.2 Suppose you want modify the Kit so that it prints out the semantic objects of the static semantics. (Semantic objects are things like structure and type names, signatures, and functor signatures.) You then edit the file `Kit/Common/Flags.sml`, which initially contains

```
(*$Flags: FLAGS*)
functor Flags(): FLAGS =
  struct
    val DEBUG_LEXING       = false
    val DEBUG_PARSING      = false
    val DEBUG_ELABTOPDEC   = false
    val DEBUG_ENVIRONMENTS = false
    val DEBUG_EVALDEC      = false
    ....
  end;
```

changing it to

```
(*$Flags: FLAGS*)
functor Flags(): FLAGS =
  struct
    val DEBUG_LEXING       = false
    val DEBUG_PARSING      = false
    val DEBUG_ELABTOPDEC   = true
    val DEBUG_ENVIRONMENTS = true
    val DEBUG_EVALDEC      = false
    ....
  end;
```

The first line `(*$Flags: FLAGS*)` is a comment for the `make` system. Inside the kit session, we now type

```
touch "Flags";
```

which tells the `make` system that we have altered the file which contains the label `Flags`. The `make` system responds with a message acknowledging that `Flags` has been altered; we then type

```
make "K";
```

which instructs the `make` system to re-compile as much of the Kit as it needs. After some time, you will see something like

```
structure K :
  sig
    val parse : unit -> unit
    val elab : unit -> unit
    val eval : unit -> unit
    val parseFile : string -> unit
    val elabFile : string -> unit
    val evalFile : string -> unit
  end

open K
   [Make: K now OK]
[closing %Make.tmp%]
val it = () : unit
-
```

You can now see the effect of the change by executing `simpleprog.sml` in the new Kit:

```
     evalFile "../examples/simpleprog.sml";
```

which prints out a lengthy piece of prose, beginning with

```
{{NAT}, 3}


structure ARITH :
   sig(3)
      NAT LAMBDA (). NAT/ Zero NAT Succ (NAT -> NAT)
      val twice: (NAT -> NAT) con Succ: (NAT -> NAT) con Zero: NAT

   end
```

Readers who are familiar with the Definition of Standard ML will find this output fairly readable. The {{`NAT`}, `3`} is a name set, containing a type and a structure name, one for the `ARITH` structure and one of the `NAT` type. A couple of lines further down, `sig(3)` shows the structure name of the `ARITH` structure. The

```
LAMBDA ().  NAT/ Zero NAT Succ (NAT -> NAT)
```

is a type structure, where `LAMBDA ().  NAT` is the type function and `Zero NAT Succ (NAT -> NAT)` is the constructor environment.

1.3.3 Making a new stand-alone program

Most ML systems have a way of creating a stand-alone program from a ML function. The method varies a bit from system to system. The `evalFile` program of Section 1.1 was created from a kit session in SML/NJ by typing

```
exportFn("evalFile",fn ((_::infile::rest),_) => evalFile infile);
```

You can also save the entire kit session as an executable file by typing

```
exportML "newkit";
```

Warning: The versions of the Kit you create with `exportFn` and `exportML` are large, typically 4 and 9 Mbytes, respectively.

Chapter 2

Overview of the Kit

In this section we give an overview of the Kit and illustrate the connection between the Definition and the Kit.

2.1 Parsing, Elaboration and Evaluation

The Definition introduces the notion of execution of ML programs as follows:[11, page 1]

> ML is an interactive language, and a *program* consists of a sequence of *top-level declarations*; the execution of each declaration modifies the top-level environment, which we call a *basis*, and reports the modification to the user.
>
> In the execution of a declaration there are three phases: *parsing*, *elaboration*, and *evaluation*. Parsing determines the grammatical form of a declaration. Elaboration, the *static* phase, determines whether it is well-typed and well-formed in other ways, and records relevant type or form information in the basis. Finally evaluation, the *dynamic* phase, determines the value of the declaration and records relevant value information in the basis.

The Kit follows the division of execution into parsing, elaboration and evaluation, although some extra phases have been introduced, for example for handling elaboration errors and resolving parsing ambiguities.

In the Kit, parsing is done by the *Parser*, elaboration is done by the *Elaborator* and evaluation is done by an *Evaluator*. Actually, there are two evaluators, one based on the *Interpreter* and based on the *Compiler*.

The Elaborator and the Interpreter-based Evaluator are direct implementations of the inference rules in the Definition, so that for every inference rule in the Definition there is a corresponding piece of code in the Kit.

The Compiler-based Evaluator provides an alternative means of evaluation. It compiles abstract syntax trees into expressions in a simpler intermediate language. This language is an untyped call-by-value lambda calculus, with constants and simple operations for constructing and destructing records, but with no pattern matching or special constructs for modules. There is also an interpreter for the lambda language. Although the compiler does not produce machine code, evaluating programs using the Compiler and the lambda language interpreter is still substantially more efficient than using just the Interpreter.

Interestingly, the two evaluators have exactly the same interface to the rest of the Kit. Thus one constructs a direct implementation of the Definition by assembling the Parser, the Elaborator and the Interpreter-based Evaluator whereas one obtains a more efficient system by replacing the Interpreter-based Evaluator by the Compiler-based Evaluator. This modularity is convenient and illustrates the general point that the Kit is supposed to be a collection of modules, that can be put together in several ways.

The basic top-level loop in the Kit is as pictured in Figure 2.1. The loop starts at the top left-hand corner with a basis B and a source program. The basis contains information about the identifiers that have previously been declared at top-level; initially, it is the *initial basis* B_0[11, Appendices C and D].

The result of executing the source program is a basis B', which is then added to the old basis B (see the bottom right-hand corner) and the loop starts all over.

In the Kit, a basis consists of three components: an *infix basis*, a *static basis*, and a *dynamic basis*. The infix basis maps identifiers to their infix status. The static basis maps identifiers to semantic objects in the static semantics — for example it maps value variables to types. Finally, the dynamic basis maps identifiers to the semantic objects of the dynamic semantics — for example it maps value variables to values.

Looking more closely at the internal structure of Figure 2.1, we see that the result of parsing a source program is a top-level declaration *topdec*, in the form of an abstract syntax tree, and a new infix basis which records the meaning of the fixity directives encountered in the source program. The abstract syntax tree is then elaborated in the old static basis, resulting in a new top-level declaration *topdec′* and a static basis, which records the static information of the identifiers declared by *topdec*. Here *topdec′* differs from *topdec* in that it has been decorated with various static information, for example information about type errors, which is dealt with in a separate phase after elaboration (not shown in the picture).

The loop in Figure 2.1 corresponds to Section 8 of the Definition, which the reader is encouraged to consult at this stage. Section 8 defines a basis as just a pair $B = (B_{\mathrm{STAT}}, B_{\mathrm{DYN}})$ of a static and a dynamic basis. In the Definition, the scope rules for fixity directives are defined in Sections 2.6 and 3.3; fixity directives are not treated as part of the static and dynamic semantics. In an implementation, where one has to spell out the effect of fixity directives operationally, it is natural to extend the basis to include an infix basis.

Section 8 of the Definition also defines what is supposed to happen in cases of abnormal termination. For example, when a elaboration error occurs, evaluation is skipped and the basis is left unchanged. Figure 2.1 only pictures successful executions, but the Kit does of course also deal with abnormal termination. Some effort has gone into producing decent error reporting. For example, elaboration always results in an abstract syntax tree whether there were any errors or not; the abstract syntax tree is decorated with information about the errors that were found and a separate phase between elaboration and evaluation runs over the abstract syntax tree and collects all the error information, edits it and prints it on the terminal.

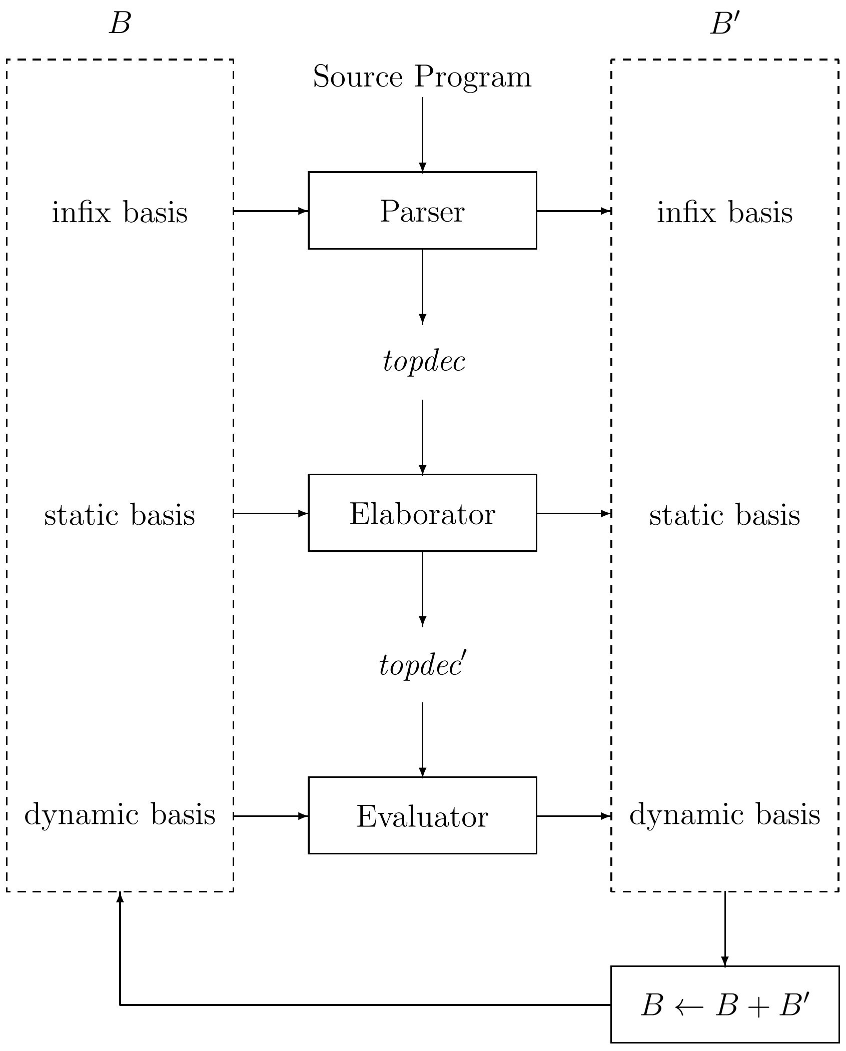

Figure 2.1: The top-level loop in the Kit

2.2 A guided tour of the Kit

In this section we shall explore some of the internal structure of the Kit. The Kit is big, so it is necessary to be able to manoeuvre around the Kit, without knowing the details of the individual modules. The purpose of this section is to provide just enough travel information that it becomes possible to move around the Kit in an informed way.

The Definition is to some extent a map of the Kit. We hope that the examples below will stimulate the use of the Definition as a "reference manual" to the Kit, and will show how closely connected the two are.

2.2.1 Where to find things

The Kit source code is found in the directory `src`. The `src` directory is shown with its major sub-directories in Figure 2.2. The `Common` directory contains modules that are used by both the Interpreter and the Compiler, for example modules concerning abstract syntax, elaboration and the top-level loop. The `Parsing` directory contains modules that are specific to parsing. The `Interpreter` directory contains modules that are specific to the Interpreter, so this is where one finds the direct implementation of the dynamic semantics of the Core and of Modules [11, Sections 6 and 7]. The `Compiler` directory contains modules that are specific to the compiler, for example the lambda language and pattern compiler. We shall now give a few samples from each: in Section 2.2.2 we give an overview of parsing, in Section 2.2.3 we give an overview of the Interpreter-based Evaluator, in Section 2.2.4 we give an overview of the Compiler-based Evaluator, in Section 2.2.5 we give an overview the Elaborator, in Section 2.2.6 we describe how all these modules are put together, and in Section 2.2.7 we finally sum up our overview with tables of the most important modules.

2.2.2 Parsing

The parser reads input (from a string or a stream) and produces an abstract syntax tree.

The parser is generated by ML-Yacc and ML-Lex.[1] Let us look at a small part of the grammar for the full Core language[11, Appendix B, page 71].

[1]ML-Yacc and ML-Lex are due to Andrew W. Appel and David R. Tardeti, based on earlier work by Nick Rothwell. The version used in the Kit is the portable version which is part of the Edinburgh SML Library, not the Standard ML of New-Jersey version.

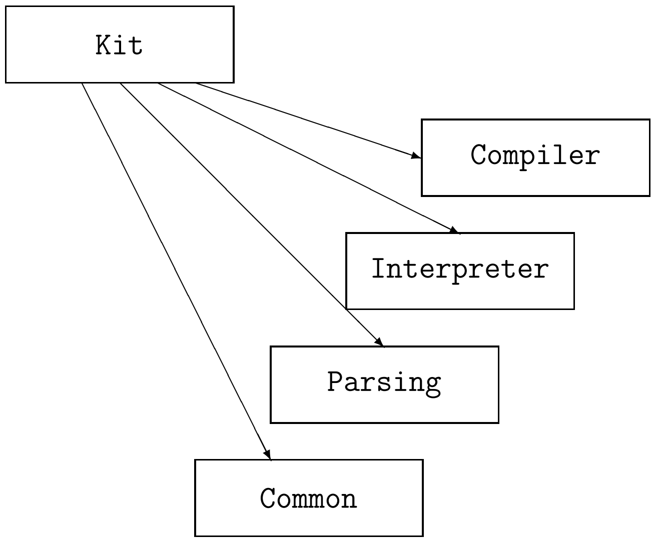

Figure 2.2: The Kit directory and its sub-directories

atexp	::=	*scon*	special constant
		⟨`op`⟩*longvar*	value variable
		⟨`op`⟩*longcon*	value constructor
		⟨`op`⟩*longexcon*	exception constructor
		`{` ⟨*exprow*⟩ `}`	record
		`#` *lab*	record selector
		`()`	0-tuple
		`(`exp_1 `,` ⋯ `,` exp_n`)`	n-tuple, $n \geq 2$
		`[`exp_1 `,` ⋯ `,` exp_n`]`	list, $n \geq 0$
		`(`exp_1 `;` ⋯ `;` exp_n`)`	sequence, $n \geq 2$
		`let` *dec* `in` exp_1 `;` ⋯ `;` exp_n `end`	local declaration, $n \geq 1$
		`(` *exp* `)`	

Here *atexp* ranges over *atomic expressions*, *exp* ranges over *expressions*, and *dec* ranges over *declarations*.

Corresponding to some of the above productions one finds the following specification to ML-Yacc in the file `Parsing/Topdec.grm` (Section 3.5.2 details the naming conventions used in the grammar):

```
AtExp:
          SCon          ( SCONatexp(PP SConleft SConright, SCon) )
        | LongOpEqIdent ( let
                            val {id, withOp} = LongOpEqIdent
                          in
                            IDENTatexp(PP LongOpEqIdentleft LongOpEqIdentright,
                                          OP_OPT(mk_LongId id, withOp)
                                         )
                             end
                        )
        | LBRACE ExpRow_opt RBRACE
                        ( RECORDatexp(PP LBRACEleft RBRACEright, ExpRow_opt) )
        | LET Dec IN Exp_ END
                        ( LETatexp(PP LETleft ENDright, Dec, Exp_) )
        | LPAREN Exp_ RPAREN
                        ( PARatexp(PP LPARENleft RPARENright, Exp_) )
        ....
```

The ML-Yacc specification associates "semantic actions" with each production. The semantic actions are written in ML, and their effect is to build an abstract syntax tree. In the production for `let` expressions, for example, `LETatexp` is a constructor defined in datatype which implements abstract syntax trees.[2]

From this grammar specification, ML-Yacc produces a parser. The parser produces abstract syntax trees according to the grammar of the Bare language[11, page 1], i.e. derived forms are expanded during parsing and are not present in the abstract syntax trees. The abstract syntax follows the figures in Sections 2.8 and 3.4 of the Definition as closely as possible. Consider the following excerpt of the grammar for the Bare language [11, Figure 3, page 8].

atexp	::=	*scon*	special constant
		⟨`op`⟩*longvar*	value variable
		⟨`op`⟩*longcon*	value constructor
		⟨`op`⟩*longexcon*	exception constructor
		`{` ⟨*exprow*⟩ `}`	record
		`let` *dec* `in` *exp* `end`	local declaration
		`(` *exp* `)`	

The abstract syntax in the Kit is found in the file `Common/DEC_GRAMMAR`:

```
signature DEC_GRAMMAR =
sig
  type lab        (* labels *)
   and scon       (* special constants *)
   and con        (* constructors *)
   and id         (* identifiers - variables or constructors *)
   and longid     (* long identifiers - variables or constructors *)
   and tyvar      (* type variables *)
```

[2]The `PP LETleft ENDright` in the same line has the effect of decorating the node in the syntax tree with position information (line and column) about the beginning and end of the phrase in the source text.

```
   and tycon      (* type constructors *)
   and longtycon  (* long type constructors *)
   and excon      (* exception constructors *)
   and longstrid  (* structure identifiers *)

  type info       (* info about the position in the source text, errors etc *)

  datatype 'a op_opt = OP_OPT of 'a * bool

  datatype atexp =
        SCONatexp of info * scon |
        IDENTatexp of info * longid op_opt |
        RECORDatexp of info * exprow Option |
        LETatexp of info * dec * exp |
        PARatexp of info * exp
  .....
```

Notice the strong similarity with the excerpt of the grammar. The specified types (`lab`, `scon`, ...) have been named according to the Definition. Notice that we even have a constructor for parenthesised expressions `PARatexp`, even though no sane compiler writer would retain parenthesis in his abstract syntax trees! Similarly, we have mindlessly recorded whether identifiers were prefixed by `op` (hence the `op_opt`, corresponding to the ⟨`op`⟩ in the grammar).

The main difference between the grammar in the Definition and the datatype specification above is that every node in the specified datatype has an `info` field, which can be used to hold information such as, for example, information about type errors and information about whether the node is the translation of a derived form.

Note that `info` is a specified type and can be matched by different declarations of `info`. Indeed this is exploited. The structure that declares the abstract syntax trees that are given as input to the Elaborator matches the above specification and so does the structure that declares the abstract syntax trees that are returned from the Elaborator, but the structures differ in (amongst other things) their declaration of the `info` type. Before elaboration, `info` contains information about position in the source text; after elaboration, it also contains information about type errors.[3]

A minor difference between the grammar in the Definition and the datatype specification above is that the `datatype` does not distinguish between variables, constructors and exception constuctors (hence the single constructor `IDENTatexp`). The reason for this deviation from the grammar is that identifier status is best resolved during elaboration,

[3]An alternative to having two different structures that match the `DEC_GRAMMAR` signature would be to make all datatypes in the abstract syntax polymorphic in the `info` type; however, this would make the datatype declaration and specifications look less elegant. Another alternative is to use a single, more general, `info` type before and after elaboration, but that is less secure: one could pass trees with the wrong `info` data to (say) the compiler and there would be no help from the type system to spot the mistake. We want the types to be distinct, and ML is good at letting us do this.

not during parsing. Notice that it is possible to use the `atexp` datatype to describe the abstract syntax tree that contains resolved identifiers simply by letting the implementation of `longid` make the distinction between the three kinds of identifiers. Thus the Elaborator accepts as input trees where `longid` is a type of unresolved identifiers and produces trees where `longid` is a type of resolved identifiers.

2.2.3 The Interpreter

The directory `Interpreter` contains the Interpreter-based Evaluator. The Interpreter is split in two, one interpreter for the Core and one interpreter for Modules. The former is a direct implementation of the dynamic semantics of Core, as defined in [11, Section 6], while the latter is a direct implementation of the dynamic semantics of Modules, as defined in [11, Section 7].

Looking first at the Core Interpreter, let us continue our trace of the execution of `let` expressions. Rule 108 in the Definition (page 51) defines the evaluation of `let` expressions as follows:

$$\frac{E \vdash \mathit{dec} \Rightarrow E' \qquad E + E' \vdash \mathit{exp} \Rightarrow v}{E \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{exp}\ \texttt{end} \Rightarrow v}$$

Here E and E' are dynamic environments[4] and the rule can be read as follows: if the declaration *dec* evaluates to E' and the body, *exp*, of the `let` expression evaluation to v in $E{+}E'$ (which means E extended with the bindings declared by *dec*), then `let` *dec* `in` *exp* `end` evaluates to v in E.

The piece of the Kit that corresponds to this inference rule is found in the file `Interpreter/EvalDec.sml`. Notice that rule 108 occurs in a section for inferring sentences of the form

$$\boxed{E \vdash \mathit{atexp} \Rightarrow v/p}$$

where v ranges over values, and p ranges over exception packets. The corresponding function in the evaluator is called `evalAtexp` and it has the type:[5]

```
val evalAtexp :  Env * atexp -> Val
```

The body of `evalAtexp` is faithful to the inference rules right down to the choice of variable names (even though this occationally conflicts with the general convention that value variables should not start with a capital letter):

```
  fun evalAtexp(E, atexp) =
    case atexp of
```

[4]The definition of dynamic environments is found in [11, Figure 13, page 47]; suffice to say that an environment contains a *variable environment*, which maps value variables to values and that this environment is used for looking up the values of free variables during evalation.

[5]Note that the result type of `evalAtexp` does not cater for exception packets. This is because exceptions and side-effects are implemented directly by ML exceptions and side-effects in the Interpreter, i.e. we have not directly followed the exception and store conventions[11, Section 6.7, page 49] that generate a large number of additional inference rules.

```
          ....

        | LETatexp(_, dec, exp) =>
            let
              val E' = evalDec(E, dec)
            in
              evalExp(C.E_plus_E(E, E'), exp)
            end
```

In many ways, the Kit is more modularised than the Definition. This is particularly true of the handling of semantic objects, such as environments and values. The Definition defines these concretely in terms of set equations. For example, variable environments are finite maps from variables to values[11, Figure 13, page 47]. The Kit, on the other hand, uses data abstraction extensively to hide the implementation of the semantic objects. Indeed, the Evaluator knows that there are such things as environments and values, and certain operations on these (such as `E_plus_E` in the above example) but it has no idea about how they are implemented. The way this abstraction is achieved is by heavy use of functors. Hence, the elaborator is a functor which is parameterised on all the structures it refers to, amongst which is also the structure which implements the semantic objects in the dynamic semantics of the Core:[6]

```
functor EvalDec(structure Grammar : DEC_GRAMMAR
                structure BasicValue: BASIC_VALUE
                structure SpecialValue: SPECIAL_VALUE
                structure CoreDynObject : CORE_DYNOBJECT
                structure ModuleDynObject : MODULE_DYNOBJECT
                  sharing type ModuleDynObject.Env = CoreDynObject.Env
                structure PP : PRETTYPRINT
                  sharing type CoreDynObject.StringTree = PP.StringTree
                structure Crash : CRASH
                ....
       ): EVALDEC =
  struct
    type dec = Grammar.dec
    type Env = CoreDynObject.Env
    type Val = CoreDynObject.Val

    structure C = CoreDynObject

    ....
```

[6]For brevity, we have omitted some of the parameters to `EvalDec` and most of the sharing constraints. When presenting excerpts of functors, we shall often omit some of the parameters and sharing constraints.

```
    fun evalAtexp(E, atexp) = .... C.E_plus_E(....)

    ....

  end;
```

Because `EvalDec` is a functor, all it knows about the dynamic objects of the Core language is what the signature `CORE_DYNOBJECT` specifies. If one inspects this signature (look in the file `Interpreter/CORE_DYNOBJECT.sml`), one sees that it just specifies types and operations on those types and does not contain any external sharing constraints with specific types or structures.

As we see from the program excerpt above, the "interface" to the evaluator for the Core is the signature `EVALDEC` (file: `Interpreter/EVALDEC.sml`)

```
        signature EVALDEC =
          sig
            type dec
            type Env

            exception UNCAUGHT of string
            val eval: Env * dec -> Env
          end;
```

which is a reasonable signature, considering that the dynamic semantics for the Core[11, Section 6, page 53] allows one to infer statements of the form

$$\boxed{E \vdash dec \Rightarrow E'/p}$$

So much for the dynamic semantics of the Core. The dynamic semantics of Modules is implemented along the same lines. This time, let us look at the code top-down. The interface to the modules evaluator is (file: `Common/EVALTOPDEC.sml`)

```
        signature EVALTOPDEC =
          sig
            type topdec
            type DynamicBasis
            type Pack

            val RE_RAISE: Pack -> unit
            exception UNCAUGHT of Pack
            val pr_Pack: Pack -> string

            val eval: DynamicBasis * topdec -> DynamicBasis
            val FAIL_USE: unit -> unit

            type StringTree
            val layoutDynamicBasis: DynamicBasis -> StringTree
          end;
```

which is a reasonable signature, considering that the Definition[11, Section 7, page 62] gives rules for inferring statements of the form

$$\boxed{B \vdash topdec \Rightarrow B'/p}$$

(The signature also corresponds nicely to the lower third of Figure 2.1.) Here and everywhere else, names with `layout` and `StringTree` have to do with pretty printing — the Kit has a pretty printer which it uses in almost every module.[7] Any exception raised in the evaluation which propagates to the top level causes the meta-exception `UNCAUGHT` to be raised with a packet. The functions `RE_RAISE` and `FAIL_USE` are part of the implementation of a `use` function (see Chapter 7).

The evaluator for the Modules is a functor `EvalTopdec`, with the above result signature `EVALTOPDEC`. `EvalTopdec` is parameterised on, among other things, a Core language evaluator; here is the skeleton of `EvalTopdec` (file: `Interpreter/EvalTopdec.sml`)

```
functor EvalTopdec(structure Grammar: TOPDEC_GRAMMAR
   structure ModuleDynObject: MODULE_DYNOBJECT
   structure CoreDynObject: CORE_DYNOBJECT
   structure EvalDec: EVALDEC
                   ....
  ): EVALTOPDEC =
  struct
   ....
  end;
```

Correponding to the places where the Modules language is built on top of the Core language, the Module evaluator calls the Core evaluator as a “subroutine”. However, because the signature `EVALDEC` gives a very limited view of the Core evaluator, the Modules evaluator and the Core evaluator are almost completely separated.

The two evaluators (the one for the Core and the one for Modules) are combined to form the Interpreter-based Evaluator, whose signature is

[7]There is also a module `Report` for assembling and indenting lines of text. This is sometimes instead of the pretty printer.

```
signature EVALUATION =
  sig
    structure DynamicBasis: DYNAMIC_BASIS

    structure EvalTopdec: EVALTOPDEC
      sharing type EvalTopdec.DynamicBasis = DynamicBasis.Basis

    structure ValPrint: VAL_PRINT
      sharing type ValPrint.DynamicBasis = DynamicBasis.Basis
  end;
```

The functor `Evaluation` now builds the evaluator using a great number of functor applications to combine all the modules that are specific to evaluation, including the Core evaluator and the Modules evaluator (see file `Interpreter/Evaluation.sml`).

2.2.4 Compiler

The directory `Compiler` contains those modules that are specific for the Compiler-based Evaluator. At present, the Compiler only works for the Core language.

The Interpreter-based Evaluator and the Compiler-based Evaluator have exactly the same interface, namely the signature `EVALUATION` shown in the previous section. This is why one can replace one by the other fairly easily.

The Compiler translates (Core) declarations into an intermediate language, called the *lambda langauge*. The lambda language is a simple call-by-value functional language, with record construction and de-construction, the ability to switch on integers and other constants (constructors have been compiled into integers), exceptions and side-effects. Its full specification is given below, not that we will explain all the details here, but we want to give an impression of how small the language is (file: `Compiler/LAMBDA_EXP.sml`). (A more detailed explanation of the lambda language is given in Section 6.5.)

```
signature LAMBDA_EXP =
  sig
    type lvar
    type (''a, 'b) map

    datatype LambdaExp =
        VAR      of lvar (* lambda variables. *)
      | INTEGER  of int (* constants... *)
      | STRING   of string
      | REAL     of real
      | FN       of lvar * LambdaExp (* function-terms. *)
      | FIX      of lvar list * LambdaExp list * LambdaExp
                                      (* mutual recursive fns. *)
      | APP      of LambdaExp * LambdaExp (* function application. *)
      | PRIM_APP of int * LambdaExp (* primitive function application. *)
      | VECTOR   of LambdaExp list (* records/tuples. *)
      | SELECT   of int * LambdaExp (* con/record indexing. *)
      | SWITCH_I of int Switch (* switch on integers. *)
      | SWITCH_S of string Switch (* ...strings *)
      | SWITCH_R of real Switch (* ...reals *)
      | RAISE    of LambdaExp (* raise exception *)
      | HANDLE   of LambdaExp * LambdaExp (* exception handling. *)
      | REF  of LambdaExp (* ref(expr) *)
      | VOID (* nil, () etc. *)
    and 'a Switch = SWITCH of {arg: LambdaExp,
       selections: ('a, LambdaExp) map,
       wildcard: LambdaExp Option
(* mandatory for REAL or STRING switches. *)
      }
    ....
    type StringTree
    val layoutLambdaExp: LambdaExp -> StringTree
  end;
```

Notice that this language neither has type declarations nor type annotations on variables. (A variable in the lambda language is called an `lvar`, which is short for "lambda variable".) Also note that this language does not have pattern matching.

A significant part of the Compiler is the *Match Compiler*.[8] The Match Compiler compiles a match into a *decision tree*, which is subsequently transformed into an expression in the lambda language.

The Match Compiler is non-trivial, and will not be described here. The rest of the Compiler is more straightforward. One minor complication is that the lambda language does not have declarations as a phrase class. Intutively, one might expect that a declaration $dec \equiv$ `val` var_1 `=` exp_1 `and` $\cdots$ `and` var_n `=` exp_n would be compiled into a lambda term of the form $[(lvar_1, e_1), \ldots (lvar_n, e_n)]$, where $lvar_1, \ldots lvar_n$ are lambda variables corresponding to $var_1, \ldots, var_n$ and $e_1, \ldots, e_n$ are the results of compiling $exp_1, \ldots, exp_n$. Instead, the compiler actually compiles dec into a pair $(CEnv, f)$, where $CEnv$ (short for "compile-time environment") would be the mapping $\{var_1 \mapsto lvar_1, \ldots, var_n \mapsto lvar_n\}$ in our example, and f is an ML function f of type LambdaExp$\rightarrow$LambdaExp, where for all e', $f(e')$ in our example evaluates to

$$\texttt{FIX}([(lvar_1, \ldots, lvar_n), [e_1, \ldots, e_n], e'))]$$

which in more conventional notation would be written

$$\texttt{let}\ lvar_1 = e_1, \ldots, lvar_n = e_n\ \texttt{in}\ e'$$

[8]A *match* in ML is a sequence of *rules*, each rule having a *pattern* on the left-hand side and an expression on the right-hand side. Thus the Match Compiler does what is often referred to in the literature as *pattern compilation*.

This explains the signature of the Core compiler (file: `Compiler/COMPILE_DEC.sml`):

```
        signature COMPILE_DEC =
          sig
            type dec
            type CEnv (* Compiler env: ident -> lvar/prim. *)
            type LambdaExp

            val compileDec: CEnv -> dec -> (CEnv * (LambdaExp -> LambdaExp))
          end;
```

The compiler itself is a functor which matches the above signature (file: `Compiler/CompileDec.sml`)

```
functor CompileDec(....
   structure Grammar: DEC_GRAMMAR
   structure Lvars: LVARS
   structure LambdaExp: LAMBDA_EXP
   structure CompilerEnv: COMPILER_ENV
   structure MatchCompiler: MATCH_COMPILER
                   ....
  ): COMPILE_DEC =
  struct
    ....

    fun compileAtexp env atexp =
      case atexp
        of SCONatexp(_, SCon.INTEGER x) => INTEGER x
         | SCONatexp(_, SCon.STRING x) => STRING x
         | SCONatexp(_, SCon.REAL x) => REAL x
         | LETatexp(_, dec, exp) =>
             let
               val (env1, f) = compileDec env (false, dec)
             in
               f(compileExp (env plus env1) exp)
             end
    ....
  end;
```

Here we have shown the compilation of `let` expressions, which is almost trivial; the boolean `false` in the recursive call indicates that `dec` is not at top-level.

The directory `Compiler` contains a functor, `Evaluation` (file: `Compiler/Evaluation.sml` which builds the Compiler-based Evaluator using a large number of functor applications.

2.2.5 Elaboration

The Elaborator is also broken into two parts: an elaborator for the Core and an elaborator for Modules. The former is an implementation of the static semantics for the Core, as defined in [11, Section 4], while the latter is an implementation of the static semantics for Modules, as defined in [11, Section 5]. The Elaborator is located in the directory `Common`, since both the Interpreter-based Evaluator and the Compiler-based Evaluator use the Elaborator. We now look at the Core and the Modules elaborators in turn.

The elaborator for the Core language is a functor. Its result signature gives the interface to the elaborator (from the file `Common/ELABDEC.sml`):

```
signature ELABDEC =
sig
  type Context and Env and Type

  type PreElabDec and PostElabDec
  type PreElabTy  and PostElabTy

  val elab_dec : Context * PreElabDec -> Env  * PostElabDec
  and elab_ty  : Context * PreElabTy  -> Type * PostElabTy
end;
```

In other words, all the rest of the Kit can do with the Core elaborator is to call either `elab_dec` or `elab_ty`. The rules for the static semantics of declarations *dec*[11, pages 25–26] allow one to infer sentences of the form

$$\boxed{C \vdash dec \Rightarrow E} \tag{2.1}$$

where C is a context and E is a static environment.[9] The type of `elab_dec` is therefore as one would expect, having seen (2.1), when one considers that the elaborator has to produce a (modified) abstract syntax tree as well as an environment.

Looking at the skeleton of the Elaborator for the Core (from the file `Common/ElabDec.sml`) we see that it is parameterised on the objects of the static semantics of the Core:

```
functor ElabDec(structure IG: DEC_GRAMMAR
                structure OG: DEC_GRAMMAR
                structure Environments: ENVIRONMENTS
                structure StatObject: STATOBJECT
                ....
       ) : ELABDEC =
  struct
   ....
  end;
```

[9]Contexts and (static) environments are defined in [11, Figure 10, page 17]; suffice to say that a static environment contains a (static) variable environment which maps value variables to type schemes and that a context contains a static environment.

Returning to our running example, the execution of `let` expressions, we find the elaboration rule (6) on page 24 of the Definition:

$$\frac{C \vdash \mathit{dec} \Rightarrow E \qquad C \oplus E \vdash \mathit{exp} \Rightarrow \tau}{C \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{exp}\ \texttt{end} \Rightarrow \tau}$$

Here τ is a *type*[11, Figure 10, page 17].

There is a fundamental difference between the inference rules in the dynamic semantics and the rules in the static semantics, and this applies both to the Core language and to Modules. Whereas the rules for evaluation essentially are deterministic, the rules for elaboration rely on non-determinism to express polymorphism (in the Core) and sharing (in the Modules). Thus it is less straightforward to translate the inference rules in the static semantics to Kit code than it is to translate the inference rules in the dynamic semantics.

The algorithm which faithfully implements the elaboration rules for the Core is based directly on Damas' and Milner's algorihm W[8]. If W is applied to an expression e and a type assignment A, $W(A, e)$ either fails (in case e is not typable in any substitution instance of A) or returns a substitution S and a type τ such that $S(A) \vdash e \Rightarrow \tau$. Here is the part of the Core Elaborator that deals with `let` expressions (from the file `Common/ElabDec.sml`):

```
  fun elab_atexp (C : Environments.Context, atexp : IG.atexp) :
     (Substitution * StatObject.Type * OG.atexp) =
    case atexp of
      ....
      (* let expression *)                                   (* rule 6 *)
    | IG.LETatexp(i, dec, exp) =>
       let
         val (S1, E, out_dec)   = elab_dec(C,dec)
         val (S2, tau, out_exp) = elab_exp((S1 onC C) C_cplus_E E, exp)
       in
        (S2 oo S1, tau, OG.LETatexp(okConv i,out_dec,out_exp))
       end
      ....
```

Here `IG` stands for input grammar, `OG` stands for output grammar, `S1` and `S2` are substitutions, `oo` is infix composition of substitutions, `onC` is infix application of a substitution to a context, and `i` is the `info` field.

We cannot get tired of stressing the importance of data abstraction obtained through the use of functors. The elaborator has no idea what substitutions are, nor does it know how composition of substitutions is implemented. It is therefore possible to use the Kit elaborator in an implementation where substitutions are done by side effects (so that type `subst` would be implemented as `unit`); in the Kit, substitutions are actually implemented in a purely applicative (and rather simple-minded) way, but the whole point is that the modules type discipline can ensure that the Evaluator does not rely on this particular representation of substitutions.

Now let us turn to the elaboration of Modules. The Modules elaborator is a functor with the following result signature (file: `Common/ELABTOPDEC.sml`)

```
signature ELABTOPDEC =
  sig
    type StaticBasis

    type PreElabTopdec and PostElabTopdec

    val elab_topdec: StaticBasis * PreElabTopdec ->
                       StaticBasis * PostElabTopdec
    type StringTree
    val layoutStaticBasis: StaticBasis -> StringTree
  end;
```

This interface is very similar to the interface to the Core elaborator, and it corresponds nicely to the middle third of Figure 2.1. In the Definition, the corresponding sentence form is[11, page 44]:

$$\boxed{B \vdash \mathit{topdec} \Rightarrow B'}$$

The Modules elaborator itself if found in the file `Common/ElabTopdec.sml`. Below we show the skeleton of the functor and the piece of code corresponding to the rule for generative structure expressions[11, rule 53, page 37][10]

$$\frac{B \vdash \mathit{strdec} \Rightarrow E \qquad m \notin (N \text{ of } B) \cup \text{names}\, E}{B \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} \Rightarrow (m, E)}$$

```
functor ElabTopdec(structure IG: TOPDEC_GRAMMAR
   structure OG: TOPDEC_GRAMMAR
   structure ElabDec: ELABDEC
   structure ModuleStatObject: MODULE_STATOBJECT
   structure ModuleEnvironments: MODULE_ENVIRONMENTS
   structure U : MODULE_UNIFY
                   ....
                ) : ELABTOPDEC  =
  struct
```

[10]This is the rule that formalises what is meant by saying that the structure expression `struct` *strdec* `end` is *generative*: the name m of the resulting structure (m, E) is chosen outside the set $(N \text{ of } B) \cup \text{names}\, E$.

```
  ....

  fun elab_strexp (B : Env.Basis, strexp : IG.strexp) :
    (Stat.Str * OG.strexp) =

    case strexp of

      (* Generative *)
      IG.STRUCTstrexp(i, strdec) =>
        let
          val (E, out_strdec) = elab_strdec(B, strdec)
          val m = Stat.freshStrName()
        in
          (Stat.mkStr(m,E), OG.STRUCTstrexp(conv i, out_strdec))
        end
  ....
end;
```

2.2.6 Putting it all together

We have now seen several examples of functors and signatures that implement different parts of the Definition. We shall now see how all these modules are put together.

The basic idea is that the Kit is built by composing functor applications, much as one composes functions. Since the Kit consists of a large number of functors, it would certainly not be practical to try to build the Kit interactively by typing in functor applications to the ML system. Instead, there are a few functors in the Kit that are *linking functors*, i.e. their job is just to apply other functors. The linking functors of the Kit appear in Figure 2.3. Notice that there are two functors called `Evaluation`, one in the `Interpreter` directory and one in the `Compiler` directory. They both match the `EVALUATION` signature (file: `Common/EVALUATION.sml`) and produce the Interpreter-based Evaluator and the Compiler-based Interpreter, respectively. The interface of the Kit at top level is found in the file `Common/KitCompiler.sml`:

```
    functor KitCompiler(val prelude: string) :
      sig
        val parse: unit -> unit
        val elab: unit -> unit
        val eval: unit -> unit

        val parseFile: string -> unit
        val elabFile: string -> unit
        val evalFile: string -> unit
```

```
          end =
        struct
            ....
        end
```

which are the 6 functions we saw in our "first session with the Kit" (Section 1).

In Figure 2.4 we show where a number of other important parts of the Kit are found.

2.2.7 Summary

The Kit is made up of a number of self-contained functors and signatures and a few modules whose job it is to apply all the functors to build the system. Almost all functors and signatures are *closed*, meaning that the only free identifiers they contain are either signature identifiers or are declared in the initial basis or the Edinburgh SML Library. We regard this extreme degree of functorisation as essential for the modularity and readability of the Kit: every functor explicitly states what it requires and with very few exceptions (such as the abstract syntax trees), the implementation of specified types is totally hidden from the functors that use it.

This style of programming makes it is fairly easy to find one's way around the Kit, we hope.[11] For every functor there is a result signature which gives the interface to the functor; very often, one does not have to look inside functors to find out what they do — look at the result signature to find out what the functor produces and look at its parameter list to see what it depends on.

Do not read functors. Read signatures.

In Figure 2.5 we list all the signatures we have looked at in this section.

Finally, a word about the naming of files, functors and signatures. The convention is that the result signature of a functor `Id` is found in a file with the name `ID.sml`, whereas the functor itself is found in `Id.sml`. In general, filenames with upper-case letters and underscores are used for signatures, whereas a mixture of cases indicates a file containing a

[11]It also forces one to write an awful lot of sharing constraints, but that is something one learns to live with.

Name	File	Builds
`Parse`	`Parsing/Parse.sml`	the parser for the whole langauge
`Evaluation`	`Interpreter/Evaluation.sml`	the Interpreter-based Evaluator
`Evaluation`	`Compiler/Evaluation.sml`	the Compiler-based Evaluator
`KitCompiler`	`Common/KitCompiler.sml`	the entire Kit, incl. the Elaborator

Figure 2.3: Linking functors

File	**Contains**
`Common/INTERPRETER.sml`	signature for the top-level loop
`Common/Interpreter.sml`	the top-level loop
`Common/PRETTYPRINT.sml`	signature for the pretty printer
`Common/CRASH.sml`	signature for handling of fatal errors
`Common/DF_INFO.sml`	signature for info about derived forms

Figure 2.4: Other files of interest

File	**Contains**
`Common/DEC_GRAMMAR.sml`	Grammar for Core declarations
`Common/TOPDEC_GRAMMAR.sml`	Grammar for Core declarations
`Common/STATOBJECT.sml`	Objects in the static semantics of the Core
`Common/ELABDEC.sml`	The elaborator for the Core
`Common/MODULE_STATOBJECT.sml`	Objects in the static semantics of Modules
`Common/ELABTOPDEC.sml`	The elaborator for Modules (and the Core)
`Interpreter/CORE_DYNOBJECT.sml`	Objects in the dynamic semantics of the Core
`Interpreter/EVALDEC.sml`	The Core Interpreter
`Interpreter/MODULE_DYNOBJECT.sml`	Objects in the dynamic semantics of Modules
`Common/EVALTOPDEC.sml`	The Modules Interpreter
`Compiler/LAMBDA_EXP.sml`	The lambda language
`Compiler/COMPILE_DEC.sml`	The Core declaration Compiler
`Common/EVALUATION.sml`	Evaluator, both Core and Modules

Figure 2.5: Important signatures, in the order: parsing, elaboration and evaluation

functor. Also, a file name with `TOPDEC` in it indicates that the file contains something that has to do with Modules, whereas a file name with `DEC` in it indicates that the file contain something to do with the Core; a file name with `PP` in it indicates that the file contains something to do with pretty printing. For example, the file `Common/PPDECGRAMMAR.sml` provides the signature for the pretty printing functions for Core abstract syntax trees.

Chapter 3

Parsing

3.1 Introduction

This chapter describes the parsing scheme of the SML Kit. We introduce the grammar datatype, or abstract syntax, and the grammar specification itself, explaining the conventions used. There are various mechanisms employed to resolve the ambiguities and parsing difficulties of SML, and we describe these together with the mechanisms which deal with infix operators, derived forms, and so on.

We use the ML-Lex[4] and ML-Yacc[5] packages. Both have been modified to produce portable code running under the SML Library[7], rather than SML/NJ-specific code, so that the resulting parser can be compiled under other SML systems. In fact, these versions of ML-Lex and ML-Yacc are themselves portable, so that all Kit development work can take place using a different SML compiler.

The generated lexing and parsing functors are supported by a number of utility functors for dealing with infixes, derived-form elimination, and so on. There are also utility functors for lexing, providing positional information for error messages, and so on. All these components are gathered together in the functor `Parse` (Section 7.3).

Section 3.2 describes the abstract syntax and its parameterisation. Section 3.3 covers lexing issues, and Section 3.4 overviews the general structure of the parser. We cover the grammar specification in Section 3.5, and the functor structure and linkage in Section 7.3. The constituent files of the parsing system are described in Section 3.7.

At this stage we should probably mention the main thing that the parsing phase does *not* do. The Definition states a number of syntactic restrictions (Section 2.9), mainly concerned with repeated identifiers of various kinds. In fact, some of these restrictions are *not* syntactic since they are impossible to enforce at parse time, and require elaboration. Specifically, repeated identifiers in value bindings and patterns cannot be detected at parse time since it is legal to repeat value and exception constructors, and these cannot be determined before elaboration. We have chosen to enforce *none* of the rules about repeated identifiers, for the sake of simplicity; it is cleaner to deal with these errors during elaboration. Also we deal with the syntactic restriction in [11, Section 2.9, third bullet, last sentence] during elaboration. The one remaining syntactic restriction involving `val rec` is truly syntactic and can be dealt with in the grammar.

3.2 The Abstract Syntax

The abstract syntax in the Kit is split into two levels: the abstract syntax of the Core and that of the Modules. This is the same division as found in the Definition, and with the same provisos: qualified identifiers and `open` declarations are part of the Core syntax.

The Core abstract syntax provides the language phrases up to the level of declarations (*dec*), and the Modules abstract syntax provides the phrases up to top-level declarations (*topdec*). There is no type representing the phrase class *program*; instead, the top-level loop of the interpreter is responsible for parsing and evaluating a sequence of top-level declarations.

The Core and Modules abstract syntax are each a set of mutually recursive datatypes, parameterised on a number of more primitive types representing constants, identifiers of various kinds, and in general the leaves of the syntax. In addition, the Modules abstract syntax is parameterised on the datatype of declarations, *dec*.

An important aspect of any SML implementation is the classification of identifiers. In SML there is no lexical distinction between variables, data constructors and exception constructors; they are all conform to the same lexical convention. In addition, it is not always possible for the parser to infer the class of an identifier from its syntactic context. The parser therefore generates an abstract syntax tree with many of the identifiers unresolved.

The later stages of the interpreter require that all identifiers be resolved. One way of doing this is to decorate the abstract syntax with an indication of the class of each ambiguous identifier. However, there is another method which requires no decoration and is also type-secure, in that the later stages of the interpreter can only ever be passed abstract syntax which has had the identifiers resolved. This is done by parameterising the abstract syntax on the type of identifiers. The abstract syntax functor is used twice: once with generic identifiers, and once with resolved identifiers (represented as a datatype). The result is two distinct abstract syntax types, one of which can have its identifier type shared with the resolved identifier datatype.

The abstract syntax tree produced by the parser is unresolved, and contains the following type identities:

$$\texttt{TopdecGrammar.id} = \texttt{DecGrammar.id} = \texttt{Basics.Ident.id}$$
$$\texttt{DecGrammar.longid} = \texttt{Basics.Ident.longid}$$

The abstract syntax produced by the elaboration is resolved, and contains the identities

$$\texttt{TopdecGrammar.id} = \texttt{DecGrammar.id} = \texttt{Basics.Var.var}$$
$$\texttt{DecGrammar.longid} = \texttt{ResIdent.longid}$$

where

```
datatype longid = LONGVAR of longvar
                | LONGCON of longcon
                | LONGEXCON of longexcon
```

Corresponding to the two levels of abstract syntax in the Kit, there are two signatures and two functors. `DEC_GRAMMAR` and `DecGrammar` provide the abstract syntax for the Core, and `TOPDEC_GRAMMAR` and `TopdecGrammar` the abstract syntax for Modules. The signatures contain the full datatype specifications (since the interfaces to the abstract syntax modules are essentially the datatypes themselves). `DecGrammar` is parameterised on the types `id` and `longid` as described above, on a type `GrammarInfo` which provides the *info* nodes on the syntax tree, and on structures which provide the leaves of the syntax tree (`LAB` for record labels, `SCON` for special constants, `CON` for data constructors, `TYVAR` for type variables, `TYCON` for type constructors, `EXCON` for exception constructors, `STRID` for structure identifiers).[1] `TopdecGrammar` is parameterised on a similar set of types (`strid`, `longstrid`, `funid`, `sigid`, `id`, `tyvar`, `tycon`, `longtycon`, `con`, `excon`, `GrammarInfo`) as well as two syntax types from the Core language (`dec` and `ty`).

The functors are applied in two places in the linking stage. In the first (functor `TopdecParsing`), the unresolved abstract syntax is built along with the parser and the pretty-printers for the syntax. In the second (`Elaboration`) the abstract syntax is built in terms of the resolved identifiers, together with the elaborator and pretty-printers for the resolved syntax.

3.3 Lexing

The lexer is built using ML-Lex. The resulting functor is parameterised on a token specification produced by ML-Yacc, and a couple of utility functors (`LexBasics` and `LexUtils`) described below. The lexical specification is fairly straightforward. The lexer carries an abstract argument (of type `LexUtils.LexArgument`) which is used to accumulate string constants and to keep track of comment depth. Comments and strings are consumed in separate lexing states (`C` and `S`).

The lexing specification does not contain keywords, since these would add complexity to the specification and would result in large lexing tables. Instead, the lexer generates generic identifiers which are translated into keywords in module `LexUtils`.

There is some complexity in the handling of the source character stream, for two reasons. Firstly, the lexing specification needs to provide a type `pos` required by ML-Lex and ML-Yacc so that the generated lexer and parser fit together, with the positional information of the tokens finding its way into the abstract syntax. Secondly, these `pos` objects have to be converted back into values suitable for printing, including the input file name, line and column number, and the actual source text. This is all done using the abstract stream package provided by `LexBasics`.

Firstly, we consider the type of positions, `pos`. A position is either `DUMMY` (a marker used for end-of-file), or is a suspension containing a function which returns the current file name, line, column and a function mapping line numbers to lines of source text. The suspension is for efficiency; the overhead of position calculation is incurred when a position is printed, rather than for each token.

[1] `DecGrammar` does not actually need to take all these as structure arguments; many could just be types.

`SourceReader` is the type of input streams. A source reader is composed of a lexing function to produce the source text and a function mapping absolute character number to position. Source readers are built using `lexFromFile` (to read source text from a file) or `lexFromString` (to read from a string).

The generated lexer has access to the absolute character position of each token as it is recognised. From these, a position object (of type `pos`) can be built for the start and end of each token. This is the linkage required between the ML-Lex-generated lexer and ML-Yacc-generated parser, and results in abstract syntax whose tokens contain positional information. This information is propagated into the info fields of the interior nodes of the abstract syntax in the actions of the grammar (Section 3.5). Since a `pos` object can be transformed into its file name, line number, column number, and line-printing function, the original source position and surrounding lines of text can be reconstituted and printed (using `printPosition` and `displaySource`) from any part of the abstract syntax.

The purpose of the module `LexBasics` is to provide source readers and the `pos` type. `LexUtils` provides operations on the lexer argument (type `LexArgument`, aliased to type `arg` in order to interface to ML-Lex and ML-Yacc), and general functions to build tokens. The lexer argument contains the current source reader, together with a record of the current comment level and accumulated characters for string literals.

Source readers are important in the implementation of the `use` function for reading and executing SML source from files. Each invocation of `use` causes a new source reader to be created for the used source file. If an error is detected in the file, then the source reader can be discarded. The implementation and treatment of `use` is in fact quite complex, and described in detail in Section 7.7.

3.4 Parsing

Syntactically, SML is a nightmare. Parts of the syntax require indefinite lookahead, so SML is certainly not LALR/1. The meaning of the symbols "`=`" and "`*`" changes according to context, and the requirement of user-definable infixes (which can include "`=`" and "`*`") makes the situation even more complex. The grammar described in the Definition is ambiguous in places (specifically, in the treatment of "`local`"), and the description of some of the derived forms is inconsistent with descriptions of the syntax elsewhere in the Definition, as described below.

Some of these ambiguities are dealt with the the grammar itself, by using new non-terminals to impose precedence in the usual way. We detail these cases in our description of the grammar in Section 3.5. The other ambiguities are addressed in the following sub-sections.

Because of the above problems (and others), it is impossible to parse SML directly without using a specially constructed parser which can carry a complex left state encompassing the infix environment and so on. Because we choose to use ML-Yacc for parsing, we are forced to adopt an LALR/1 grammar which accepts a clean superset of SML, and then remove illegal constructions either in the rule actions or in a separate post-pass.

In passing, we should point out a couple of inconsistencies in the description of SML's syntax in the Definition. The first is minor, and involves the documentation of derived forms of the Core, as given in Appendix A, figures 15 and 16. We consider the derived forms "`()`", "`#`*lab*" and `[`exp_1`,` ... exp_n`]` to be atomic expressions, even if their equivalent forms are not. (Such equivalent forms are considered parenthesised.) This agrees with the full grammar in figure 19.

The second ambiguity involves `val rec` declarations. The Definition states that the right hand side of a `val rec` must be of the form "`fn` *match*", possibly constrained by one or more type expressions (Section 2.9, page 9). This is inconsistent: it is not possible to constrain an entire `fn` expression without bracketting it. We have chosen to allow bare `fn` expressions and constrained `fn` expressions which are contained within a single pair of brackets; this seems to be the minimal syntax which includes the cases described and which also makes sense.

3.4.1 Infixes and the Post-Pass

Because the grammar presented to ML-Yacc is fixed, it cannot deal with SML's facility to change the infix status or precedence of arbitrary identifiers, especially since such changes are subject to scoping rules. The situation is complicated by the context-dependent meaning of the symbols "`*`" and "`=`", both of which can have their infix status changed. Our only choice is to resolve infixes after they have been parsed. Rather than maintain an infix environment during the parsing process, and tackle infixes in the rule actions, it is easiest to resolve infixes after the top-level parse has completed.

The superset language accepted by the main parser contains the rules[2]

$$\begin{aligned}&\mathit{Pat} ::= \mathit{AtPat}^{+}\\&\mathit{Exp} ::= \mathit{AtExp}^{+}\end{aligned}$$

so that a pattern can comprise a sequence of one or more atomic patterns, and an expression can be one or more atomic expressions. The post-pass either resolves these phrases (`UNRES_INFIXexp`, `UNRES_INFIXpat`), or reports a syntax error if the actual input is not legal SML. The post-pass removes all `UNRES_INFIXexp` and `UNRES_INFIXpat` nodes, replacing them with "proper" expressions containing the operators.

3.4.2 Lookahead ambiguities

The ambiguities are as follows. Firstly, consider the following two (legal) phrases (which are both patterns):

id `:` *ty* `as` *pat*
atpat `:` *ty*

If the parser encounters an atomic pattern followed by "`:`", it is not possible to ascertain whether the phrase is legal without parsing the "`:`" and the following type expression. If

[2]We are using a richer metagrammar than that in the Definition for conciseness in some of the later examples. Superscripts $*$, $+$ and $?$ denote zero or more, one or more, and optional phrases, respectively.

the type is followed by "`as`" then the original atomic pattern must be an *id* (optionally preceeded by "`op`"), otherwise it may be any atomic pattern.

This problem is dealt with in a similar way to patterns containing infixes. The parser accepts a superset of SML syntax, by allowing the form

pat (`as` *pat*)$^?$

This accepts

pat `:` *ty*

in isolation, and will also pass

pat `as` *pat*

through to be transformed into

id `:` *ty* `as` *pat*

or rejected. Unlike infix analysis, this can be done during the parsing stage, in the action for the appropriate *pat* rule.

The second ambiguity involves "`fun`" declarations. Consider the phrase

`fun (`pat_1 $infix_1$ pat_2`)` $infix_2$ pat_3 `=` ...

The phrase `(`pat_1 $infix_1$ pat_2`)` has two interpretations. If it is followed by an $infix_2$, then it is the first argument of the definition of a function called $infix_2$. If not, the function being defined is $infix_1$.

`Fun` declarations are dealt with using a grammar superset which is resolved (or rejected) in a post-pass, similar to the technique used for infix patterns and expressions. One reason for this is that `fun` declarations need to be analysed in the context of an infix environment anyway. The Kit grammar accepts function bindings of the form

`fun` $atpat^+$ (`:` *ty*)$^?$ `=` *exp*

which result in `UNRES_FUNdec` nodes in the abstract syntax. These are replaced by the equivalent `val`-bindings in the post-pass.

3.4.3 Differences Between the SML Syntax Definition and the Abstract Syntax

To summarise, the main differences between the SML syntax described in the Definition (figure 19), and the abstract syntax of the Kit, are:

- Identifiers are often just of type `id` or `longid` rather than `var`, `longcon`, etc. This is because the parser cannot in general resolve identifier status. Where status can be resolved (for example, in `datatype` declarations), it is done so.

- The Kit syntax retains the "`op`" keyword. This is needed to resolve infixes in patterns, expressions and `fun`-bindings, and is retained to aid printing (rather than being folded into the *info* nodes, which is another possibility but would add complexity). The Definition considers "`op`" to be purely syntactic, and it does not occur in any of the semantic rules.
- The abstract syntax datatype contains declaration constructors `INFIX`, `INFIXR` and `NONFIX`. These are needed for the infix-resolving post-pass. Again, the Definition considers these declarations to be purely syntactic.
- The abstract syntax datatype contains the expression constructor `UNRES_INFIXexp`, the pattern constructor `UNRES_INFIXpat`, and the declaration constructor `UNRES_FUNdec`. These are generated by the parser but eliminated by the parsing post-pass, so that the abstract syntax passed to the elaborator and the later stages of the Kit will never contain `UNRES` syntax.

3.5 Grammar

We now describe the grammar of SML used by the Kit, as opposed to the abstract syntax datatype. The two do not correspond directly; for example, the grammar accepts the derived forms, although there are no constructs for them in the abstract syntax. The main purpose of this section is to present the conventions adopted in the grammar, and to describe the less obvious features of it.

3.5.1 Tokens

There are some peculiarities in the tokens (terminal symbols) adopted by the grammar. Firstly, qualified identifiers are recognised as individual `QUAL_ID` tokens. (In fact, this is not peculiar at all, and is what the Definition suggests, but other ML compilers, specifically SML/NJ, lex qualified identifiers as a series of tokens, which is illegal.) Secondly, the grammar distinguishes between single digits, positive integers of two or more digits, and negative (signed) integers. This is because there are contexts in which single digits are required (`infix` and `infixr` declarations), and in which positive integers are required (record labels).

The treatment of identifiers is rather complicated. The syntax of SML requires unqualified identifiers in some contexts (`infix` declarations and record labels, for example) while allowing qualified identifiers elsewhere. The symbol "`*`" is allowed as an identifier except in type expressions, where it serves a specific purpose. In other words, "`*`" is disallowed as a type constructor; in particular, qualified type constructors differ from other qualified identifiers in that they cannot end in "`.*`". The symbol "`=`" is a reserved word which is required to serve as an identifier, except in contexts where that would be ambiguous. We allow qualified use of "`=`" in all contexts. In addition, we syntactically allow redefinition of "`=`", since our interpreter builds the equality predicate in terms of a predefined pervasive function. This facility is disabled for user programs in the semantic analysis stage.

To get the correct treatment of identifiers, we classify them into a number of distinct nonterminals, as follows:

`Ident`**:** Any valid unqualified ML identifier, including "`*`" but excluding "`=`".

`OpIdent`**:** An identifier, as above, optionally preceeded by "`op`".

`EqIdent`**:** An identifier, as above, or "`=`".

`TypeIdent`**:** An identifier, as above, but omitting "`*`".

`LongIdent`**:** An identifier or long (qualified) identifier. This includes qualified identifiers such as "`S.*`" and "`*.S`" (which are legal), as well as qualified identifiers ending in "`=`" (since we may wish to provide the equality predicate within a structure, even though it is not legal for programs to rebind it).

`LongTypeIdent`**:** A `LongIdent` excluding "`*`" and qualified identifiers ending with "`.*`".

`LongOpIdent`**:** A `LongIdent` optionally preceeded by "`op`".

`LongOpEqIdent`**:** A `LongOpIdent` or "`=`" or "`op =`".

`Label`**:** A record label, which is either an identifier (including "`*`" but excluding "`=`"), or a positive integer of indefinite size and without leading zeros.

These classes of identifier are nonterminals, built from a set of terminal symbols which includes unqualified identifiers (`ID`) and qualified identifiers (`FULL_QUAL_ID`, `TYPE_QUAL_ID`), and which treats both "`*`" and "`=`" as reserved words. (According to the Definition, "`*`" is not reserved, although its use and meaning vary according to context.)

Because of the large number of identifier classes in the grammar, we choose not to distinguish between the different *semantic* classes of identifier (*var*, *con*, *excon*), even in contexts where identifier status is known at parse time. This is to avoid having an even larger set of identifier nonterminals. Identifiers encountered in resolvable contexts are converted into the correct semantic objects in the rule actions, so that the abstract syntax has identifiers resolved in all the cases where it is possible to do so.

3.5.2 Naming Conventions

There are some general naming conventions in the grammar. Terminals are in upper case, nonterminals in mixed case. Note the trailing underscores in the nonterminals for expressions (`Exp_`) and matches (`Match_`) to differentiate the identifiers from the pervasive exception constructors.

The Definition has a number of conventions when describing the syntax of SML. Optional phrases are enclosed in braces (< >), there is a convention for sequences of phrases, and alternative forms are listed in order of decreasing precedence. We discuss the issue of precedence later.

Optional phrases in the grammatical rules are reflected directly in the abstract syntax of the Kit and in the ML-Yacc specification. Corresponding to each optional phrase of some class t the abstract syntax contains a field of type t `Option`, and the grammar contains an explicit "option" nonterminal, named according to the optional phrase it contains and the terminal symbol (if any) that is used to introduce it. Sequenced phrases (with syntax class Xseq for some X) are implemented with specific nonterminals to deal with the sequencing. More specifically, the conventions are as follows:

- Nonterminals ending in "`_opt`" are optional phrases. Many (such as `BarConDesc_opt` and `AndDatDesc_opt`) are introduced by keywords (`BAR` and `AND` in these cases); some (such as `SigDec_opt`) are not.
- Sequences of items are represented by nonterminals ending in "`_seq0`" (zero or more items), "`_seq1`" (one or more items), "`_seq2`" (two or more) and so on. Some (such as `ExpComma_seq0` and `LongIdentEq_seq2`) are named to reflect the tokens which separate the items; others (such as `AtExp_seq1`) have no separator.
- `TyVarSeq` is a special case, being a sequenced construction as described in the Definition. The Definition also uses the construction *tyseq*; we do not, since it would introduce an ambiguity into the grammar (indefinite lookahead would be required to disambiguate type constructions from parenthesised type expressions).

3.5.3 Ambiguities resolved in the grammar

The grammar has to resolve the ambiguities present in the rules of the syntactic description contained in the Definition. Many of these ambiguities are matters of precedence, covered by the Definition's conventions in the presentation of the rules. There is, however, one remaining ambiguity: a phrase of the form

`local` *dec* `in` *dec* `end`

can be interpreted either at the Modules level (`local` over two *strdec*'s which happen to be Core declarations) or the Core level (a single *strdec* which happens to be a Core declaration containing `local`). In addition, treatment of semicolons is problematic since we wish to parse

dec `;` *dec*

as two separate phrases at top-level but as a single phrase when embedded in an enclosing syntactic construct.

To solve all these problems, the grammar is layered in a fairly obvious manner. Nonterminals with names of the form `XXX_sans_YYY` represent ambiguity-resolving intermediate phrases. Top-level declarations using `local` are regarded as modules constructs, in order to avoid indefinite lookahead when resolving phrases such as

`local` *dec* `in structure` *strbind* `end`

The various nonterminals occur as follows:

`OneDec_or_SEMICOLON`: A single core-level declaration or a semicolon;

`NonEmptyDec`: A non-empty core-level declaration, possibly containing semicolons;

`OneDec`: A single declaration, no semicolons;

`OneDec_sans_LOCAL`: Single Core-level declaration phrase, excluding the `local` form;

`StrDec_sans_SEMICOLON`: A structure declaration with no semicolons at the outermost level;

`SigDec_sans_SEMICOLON`: As above, for signature declarations;

`FunDec_sans_SEMICOLON`: As above, for functor declarations;

`Ty_sans_STAR`: Type expression without "`*`" at the outermost level.

The various `_sans_LOCAL` declaration forms eliminate the ambiguity of `local` by consigning it to the Modules level or pushing it down into an unambiguous context. Semicolons are treated similarly so that the parser will deal with them properly in nested constructs but not shift past them at top-level. `Ty_sans_STAR` is a disambiguating phrase class for the "`*`" operator in type expressions.

Semicolon separators (in phrase classes *dec*, *strdec* and *spec*) are a particularly nasty case, since they are optional and they separate optional (i.e. possibly empty) phrases. We deal with this problem by giving "`;`" the same status as the phrase class it serves to separate. The "`compose`" functions from `GrammarUtils` are there to swallow empty abstract syntax phrases which can result from this treatment. One side-effect of this treatment is that it is not possible to reconstitute semicolons when reconstituting source text from the abstract syntax (since that would require an abstract syntax class for them at least).

3.6 Linking

The various modules of the parsing system, including the ML-Lex-generated lexer, the ML-Yacc-generated parser, and the support modules, are linked into a single functor, `Parse`, with the signature shown in figure 3.1. The interface has been kept as simple as possible: parsing is done from a source of input text held in a string or a file, and in the context of the current top-level infix basis. The result of the parse, if it succeeds, is the abstract syntax for a top-level declaration, plus any top-level infix basis established by this declaration.

The reason for the intermediate type `State` is to handle parsing errors elegantly. In order to deal with error recovery properly at the (interactive) top level, source readers operate on a line-by-line basis so that the remainder of an input line can be discarded if an error is detected. An interactive source reader is created by `Parse.sourceFromStdIn`. If a phrase parses and executes successfully, the remainder of the line must be retained in case it contains additional phrases. Hence, `Parse.begin` returns a `State` which can

```
signature PARSE =
   sig
      type InfixBasis
      type topdec
      type SourceReader

      val nameOf: SourceReader -> string

      val sourceFromStdIn: unit -> SourceReader
      val sourceFromFile: string -> SourceReader
      val sourceFromString: string -> SourceReader

      type Report

      type State
      datatype Result = SUCCESS of InfixBasis * topdec * State
                      | ERROR of Report
                      | LEGAL_EOF

      val begin: SourceReader -> State
      val parse: InfixBasis * State -> Result
   end
```

Figure 3.1: The signature `PARSE`

be used repeatedly by `Parse.parse` to read phrases. If the current input line needs to be discarded, the state is discarded and a new source reader created using `sourceFromStdIn` again.

We use portable versions of the ML-Lex and ML-Yacc packages which compile under Standard ML with the Edinburgh Library (the packages bundled with SML/NJ are nonportable), and our versions generate portable code (the SML/NJ ones do not). In addition, there are various support modules for ML-Yacc, including a generic parsing engine. We use a portable version of these with the "abstractions" removed and outermost structures replaced by functors.

The interfaces and linkage of ML-Lex and ML-Yacc are messy, and we are not going to document their operation here. Our `Parse` functor links the generated files in the obvious way. `Parse` contains functor applications for the other support modules, and the top-level success and failure handling routines.

3.7 Files

The abstract syntax modules are in the `Common/` subdirectory because they are required by the entire Kit. The other parsing-related files are in the `Parsing/` subdirectory. The relevant files are as follows:

`Common/DEC_GRAMMAR.sml, Common/DecGrammar.sml:` The signature and functor for the abstract syntax datatype of the Core.

`Common/TOPDEC_GRAMMAR.sml, Common/TopdecGrammar.sml:` The signature and functor for the abstract syntax datatype of the Modules.

`Parsing/Topdec.lex:` The lexical specification (ML-Lex source file) for SML.

`Parsing/Topdec.grm:` The grammar (ML-Yacc source file) for SML, as described in Section 3.5.

`Parsing/LEX_BASICS.sml, Parsing/LexBasics.sml:` The module which provides the machinery for source readers (Section 3.3) and source position annotation and printing.

`Parsing/LEX_UTILS.sml, Parsing/LexUtils.sml:` General lexing utilities: accumulation of literal strings, construction of integer and real constants, and comment nesting. `LexUtils` provides the lexer argument (type `LexArgument`) and operations on it for string and comment accumulation.

`Parsing/Topdec.lex.sml:` The ML-Lex-generated lexer.

`Parsing/Topdec.grm.sig:` Signature for the ML-Yacc output; contains the token (terminal symbol) specifications.

`Parsing/Topdec.grm.sml:` The ML-Yacc-generated parser.

`Parsing/GRAMMAR_UTILS.sml, Parsing/GrammarUtils.sml:` Utility module for the parser. Its main purpose is to expand the derived forms; the expansion functions are called from the actions in the grammar rules. It also brings together the functions for turning strings into the various kinds of identifiers (`con`, `excon`, `lab` and so on), does special constant construction, and provides translation from positional information (`LexBasics.pos`) to *info*.

`Parsing/PARSE.sml, Parsing/Parse.sml:` The top-level of the parser. `PARSE` provides the interface to the rest of the Kit. `Parse` performs all the functor applications and links all the subsidiary modules, and provides the top-level exception handling and error reporting for the parser.

`Parsing/MyBase.sml:` Library files for the generated lexer and parser. A portable, functorised, version of the modules provided with ML-Lex and ML-Yacc.

`Parsing/INFIXING.sml, Parsing/Infixing.sml:` Infix resolution. Resolves the `infix` declarations for a top-level declaration, and expands `fun` bindings into their equivalent (underived) form.

`Parsing/INFIX_STACK.sml, Parsing/InfixStack.sml:`
Internal implementation module for `Infixing`; provides generic analysis and resolution of infix constructions for both patterns and expressions.

`Parsing/HOOKS.sml:` Calls to `use` for the generated lexer and parser, since the generated files cannot carry `Make` tags.

Chapter 4

Core Elaboration

4.1 Introduction

This section describes the Core Elaborator of the ML Kit. We give an overview of the Elaborator, and explain the more intricate parts in more detail; in particular overloading of arithmetic operators and flexible records. We also describe the implementation of the static objects and operations used in the elaborator.

We assume the reader is familiar with algorithm W [8].

The outline of the chapter is as follows. Section 5.2 gives an overview of the Elaborator, and describes how elaboration errors are handled, the implementation of syntactic restrictions, and how we determine identifier status. Moreover we describe how we resolve overloading, and the implementation of recursive value bindings is explained. Finally some miscellaneous comments are given in order to ease studying the Elaborator. Section 4.3 describe the implementation of the semantic objects and operations corresponding to [11, Section 4.1–4.9]. The constituent files of the Core Elaborator are described in Section 5.20.

4.2 Elaboration

Functor `ElabDec`, located in the file `/Common/ElabDec.sml`, implements the elaborator for the Core, i.e., it implements the static semantics for the Core as defined in [11, Section 4]. Recall from Section 2.2.5 that functor `ElabDec` is parameterised on the objects of the static semantics of the Core and that it returns a structure which can be constrained by signature `ELABDEC`.

The Elaborator does not check that patterns are irredundant and exhaustive [11, Section 4.11] as this is best done during pattern compilation.

The rules defining the static semantics in the Definition are non-deterministic. Non-determinism is used to express polymorphism. Because of this non-determinism, the rules cannot be implemented directly. Hence we need an algorithm which resolves the non-determinism. Moreover we are interested in an algorithm with the properties that if it succeeds, it returns the principal type and if the expression to be elaborated cannot elaborate according to the static semantics, then the algorithm fails. One such algorithm is Damas' and Milner's algorithm W[8], which our implementation is based directly upon.

Recall the part of the Core Elaborator that deals with `let` expressions from Section 2.2.5 — the implementation contains one function `elab_`*syn* for every phrase class *syn*.

A *dec* must elaborate to an environment (c.f. rules 17–25) so the implementation returns a principal environment instead of a principal type.

4.2.1 Elaboration errors

If an error occurs some time during elaboration of a subexpression, we insert error information into the node of the syntax tree for the subexpression (a number of utility functions with the suffix "`Conv`" in their names are used to insert this error information.) We want to continue elaboration, so that other possible elaboration errors can be found during the same run. We therefore assign a bogus type to the subexpression. To avoid propagation of an elaboration error, this bogus type is chosen to be a fresh type variable, such that no constraints are imposed by the subexpression in the rest of the elaboration.

Recall the functionality of the elaboration function `elab_dec` — besides returning an environment it returns a modified syntax tree. In other words, it is the responsibility of the user of the Core Elaborator to do whatever he or she wants with the inserted error information — the Elaborator in itself does not do anything besides recording the errors in the syntax tree.

4.2.2 Syntactic Restrictions

The Definition states [11, Section 2.9] a number of syntactic restrictions, and in Chapter 3 it is stated that some of these should be dealt with during elaboration.

Some of the comments to the rules in the Definition [11, Section 4.10] states that some property is ensured by the syntactic restrictions, see for example the comment to rule 30. If we carefully determine all properties only ensured by the syntactic restrictions, we can implement the syntactic restrictions by checking whether the properties holds. For instance, the syntactic restriction that no *conbind* may bind the same identifier twice is implemented by checking, in function `elab_conbind`, that the constructor *con* is not in the domain of the constructor environment *CE*.

Let us add the following comments to some of the rules in the Definition[1].

Rule	*Comment*
8	The syntactic restrictions ensure $lab \notin \mathrm{Dom}\varrho$.
31	By the syntactic restrictions $excon \notin \mathrm{Dom}EE$.
32	By the syntactic restrictions $excon \notin \mathrm{Dom}EE$.
41	The syntactic restrictions ensure $lab \notin \mathrm{Dom}\varrho$.

Then the comments to the rules in the Definition tells us exactly which properties to check and also where to check them.

If a property at some point does not hold, error information telling which syntactic restriction has been violated, is inserted into the syntax tree.

[1]Notice, by the way, that the side condition on rule 41 is superfluous — it is ensured by the syntactic restrictions.

4.2.3 Identifier Status

As mentioned in Section 2.2.2 and in Section 3.2 the elaborator resolves identifier status, as it cannot be done at parsing time, since there is no lexical distinction between variables, data constructors and exception constructors. The classification is performed as described below.

The compound semantic object variable environment is defined in Figure 10 of the Definition. The implementation of the variable environment (a datatype `VarEnv` defined in functor `Environments`, see section 4.3 below), has as its domain unresolved identifiers. When a *datbind* or an *exbind* is elaborated, the environment must bind the data constructors, resp. exception constructors, to their respective types (c.f. rule 30 and 31.) However, in the implementation we bind the constructors to tagged types, i.e., the range of the variable environment is implemented as the following datatype

```
datatype VarEnvRange =
    LONGVAR   of TypeScheme
  | LONGCON   of TypeScheme
  | LONGEXCON of Type
```

We will later comment on this datatype; for now the important thing to notice is the three constructors, which are used as tags. This enables us to determine the class of all applied occurrences of unresolved identifiers by looking up the identifier in the variable environment. Variables are bound to a tagged type when elaborating an atomic pattern — if the unresolved identifier is not bound to a data constructor or to an exception constructor in the environment, then the identifier must be a variable. Notice that the unresolved identifiers correspond to the nonterminals *longvar*, *longcon* and *longexcon* in the grammar in the Definition[2].

4.2.4 Overloading

The Elaborator has to resolve the overloading caused by the arithmetic operators and the flexible records (a flexible record is a record in which the *wildcard pattern row* (...) occurs.) In the Commentary [10, Page 54] it is defined precisely what it means that overloading is resolved. It is only required that overloading be resolved in each *topdec*, but implementors may choose it to be resolved in some smaller textual unit. We have chosen to require overloading to be resolved when a *dec* is parsed as a *strdec*. Thus when function `elab_dec` is called, it is able to return a principal environment (by Theorem 5.2 in the Commentary.)

[2]Actually one knows at parsing time that it must be an *longexcon* which occurs in an *exbind* — as the grammar leaves no other possibilities — but the identifier is not parsed as such, because it would be inconsistent to do so.

Arithmetic operators

In this section we shortly describe the implementation of overloaded arithmetic operators.

The overloaded operators are `abs`, `~`, `*`, `+`, `-`, `<`, `>`, `<=` and `>=`.

The general idea is the following. Initially we bind too general a type scheme to the overloaded operators. This type scheme is $\forall\alpha : \alpha \to \alpha$ (for `abs` and $\sim$) or $\forall\alpha : \alpha \times \alpha \to \alpha$ (for `*`, `+`, `-`, `<`, `>`, `<=` and `>=`.) Each time a generic instance of the type scheme is made, the type variable, which α is instantiated to, is recorded in the syntax tree (in the `info` field.) When the elaboration of a *dec* has finished, yielding a substitution S, we traverse the syntax tree, applying S on every such recorded type variable. If this results in the type `int` or the type `real`, overloading has been resolved, and we record the resulting type in the syntax tree (for use in the evaluation). Otherwise overloading cannot be resolved and error-information is inserted instead. The reason we choose to require overloading to be resolved at this point is that it is the last place where we have a hold on the substitution, and doing it this way keeps a nice separation between Core and Modules elaboration.

When a generic instance is made, we need to know whether to record a type variable or not. This should only be done for overloaded type variables. Hence we add an attribute to every type variable telling whether it is an overloaded type variable or not.

Closure

An overloaded type variable stands either for `int` or for `real`. Therefore it makes no sense to allow quantification of overloaded type variables, whence it is prohibited.

Unification

Overloaded type variables are only allowed to be unified with type `int`, `real` or other type variables. Allowing unification with, for instance, a functional type, would make it impossible later, when applying the substitution S, to get int or real. Unification with other type variables is allowed. As a result it is possible to type check the example at the top of page 55 in the Commentary.

When unifying an overloaded type variable α with another type variable β, the overloading attribute must be propagated from α to β to avoid later quantification of β.

Unification of an overloaded type variable with an explicit type variable is prohibited: an explicit type variable must elaborate to itself (Definition rule 47) whereas an overloaded type variable stands for `int` or `real`.

Initial Environment

In our scheme, overloaded variables are initially bound to a type scheme containing an overloaded type variable. To avoid letting the user know of overloaded type variables, we construct type schemes for the overloaded variables explicitly (see the file `Common/Environments.sml` and the description in Section 4.3 below) — as opposed to using the `prim` mechanism described in Section 7.5. In other words, the basic values include both `prim` and the overloaded variables.

Flexible records

We elaborate flexible records using a type inference machinery developed by Rémy [13][12] for an extension of ML with new operations on records. Actually it would have been sufficient with a simpler type inference system (more about this later.) In the following sections we will first explain Rémy's extension (readers familiar with Rémy's work may skip this section) and then describe our implementation.

Recall that a record type is defined in the Definition to be a finite map from labels to types, and that the wildcard pattern row $(\ldots)$, by rule 40, is allowed to have *any* record type ρ.

Rémy's Type Inference for Records in an Extension of ML

This section is based on [12] and [13]. In [12] Rémy describes type inference for records with operations including the empty record, selection of a field, extension of a record with a new field and field removal, but excluding the concatenation operation (which given two records is supposed to return their concatenation.)[3] In [13] he extends the earlier work to include the concatenation operation. In this paper we will only describe those ideas of Rémy which we use. For concreteness, the type system will be described for a little language.

Syntax

The language of expressions, ranged over by M, is defined by the syntax

$$\begin{array}{lll} M ::= & x & \text{variable} \\ \mid & \lambda x.M & \text{lambda abstraction} \\ \mid & M_1 M_2 & \text{application} \\ \mid & \texttt{let}\ x = M_1\ \texttt{in}\ M_2 & \text{letexpression} \\ \mid & C & \text{constant} \end{array}$$

where x ranges over a set of program variables and C ranges over a set of constants including the following operations on records.

$\{\}$	empty record
$\#lab$	field selection

where *lab* ranges over the syntactic category Lab as in the Definition.

Static Semantics

The language of types is defined by the following grammar

$$\begin{array}{rcll} \tau & ::= & t \mid \alpha \mid \tau \to \tau \mid \{\rho\} & \text{types} \\ \rho & ::= & \chi \mid abs \mid lab \mapsto \varphi, \rho & \text{rows} \\ \varphi & ::= & \theta \mid absent \mid present(\tau) & \text{fields} \end{array}$$

[3]Notice it *is* an extension of ML, as in ML one cannot define general extension or removal operations.

where α ranges over a set of type variables, χ ranges over a set of *row variables*, θ ranges over a set of *field variables* and t ranges over a set of nullary type constructors $\{int, bool, \ldots\}$.

It must hold as an invariant that the labels in a row are disjoint.

Types are equal modulo the following equations (referred to later as just "the equations").

$$\begin{array}{lcll} \{lab \mapsto \theta, lab' \mapsto \theta', \chi\} & = & \{lab' \mapsto \theta', lab \mapsto \theta, \chi\} & \text{Left commutativity} \\ \{abs\} & = & \{lab \mapsto absent, abs\} & \text{Distributivity} \end{array}$$

A template row is either *abs* or a row variable. Any field can be *extracted* from a template row using substitution if the template is a variable or distributivity if it is *abs*. The use of left commutativity should be obvious. An example shows the use of distributivity (for readability we use Standard ML syntax).

The following function

```
fun f {l = v1} = v1 + 1
```

will get the type $\{l \mapsto present(int), abs\} \to int$. When elaborating the application `f l' = 3`, the types $\{l \mapsto present(int), abs\}$ and $\{l' \mapsto present(int), abs\}$ are to be unified. To recall, two record types ϱ_1 and ϱ_2 are unifiable if they have the same domain of labels and for every label l in the domain, the type bound to l in ϱ_1 is unifiable with the type bound to l in ϱ_2. The domains of the two record types under consideration are not equal, but the distributivity rule allows us to extract a field l' in the first type and a field l in the second type, so that the domains become equal. Thus we now have to unify $\{l \mapsto absent, l' \mapsto present(int), abs\}$ and $\{l \mapsto present(int), l' \mapsto absent, abs\}$. As the field types *absent* and $present(int)$ are not unifiable, unification fails; hence the application is not accepted.

The field *absent* is used in a type for a record r to express that some field must not exist in r. Field variables are used to express that it does not matter whether or not the field is present.

The typing of the primitive operations is as follows.

$$\begin{array}{rl} \{\} & \{abs\} \\ \texttt{\#}lab & \{lab \mapsto present(\alpha), \chi\} \to \alpha \end{array}$$

The typing rules are the same as in Standard ML. One does not need to change the elaboration algorithm, one just needs to let the initial type environment contain the above shown types for the primitive operations, and then extend the unification algorithm — now there must be three kinds of substitutions (one for each kind of variable) and type equality is taken modulo the equations. As in Standard ML, principal types exists (in type schemes, Rémy quantifies over all three kinds of variables).

Implementation of Flexible Records using Rémy's Record Typing Discipline

The idea is analogous to the idea used for the arithmetic operators. During elaboration of a declaration *dec*, every occurrence of the wildcard pattern row is given a type $\{\chi\}$, where

χ is a fresh row variable, and this row variable is recorded in the syntax tree. Remember that the SML selector `#lab` is a derived form, the equivalent form being `fn {lab = var, ...} => var` so it is enough to consider what to do about the wildcard pattern row. When the elaboration of a *dec* has finished, yielding a substitution S, we traverse the syntax tree, applying S on every such recorded row variable. If this results in a type with no row or field variables, overloading has been resolved. Otherwise overloading cannot be resolved and error-information is inserted instead. The reason we choose to require overloading to be resolved at this point is the same as for the arithmetic operators.

Thus we can elaborate the following declaration.

```
let
  fun eq r1 r2 = (#l1 r1) = (#l1 r2)
  val t = eq {l1 = 1, l2 = 3}
             {l1 = 1, l3 = 3}
in
  t
end
```

We remark that this solution is more powerful than the one used in the New Jersey compiler. For instance, the New Jersey compiler does not accept the above declaration.

In the context of Standard ML, one could perhaps make do with a slightly more restricted version of Rémy 's types. For example, having absent fields and field variables does not seem necessary. However, we have preferred taking over the published system virtually unchanged as opposed to trying to specialize it and perhaps introduce errors in the process.

Implementation of types

The record type defined by Rémy is implemented directly by a datatype `RecType` in functor `StatObject`. In this functor there is a function `emptyFlexRecType` which returns the above mentioned fresh row variable needed for the wildcard pattern row.

Unification

The unification of types, is implemented in functor `StatObject`. The labels of a record type are kept sorted. When unifying two records r_1 and r_2, if both records begin with a field the two fields labelled by l_1 respectively l_2 are unified if the labels are equal. Otherwise, assuming label l_1 is less than l_2, we extract a field labelled l_1 from r_2 using either the distributivity rule (if r_2 ends with *abs*) or by unification of the row variable in r_2 (if r_2 ends with a row variable.) Let S be the identity substitution if we used the distributivity rule, otherwise the substitution mapping the row variable of r_2 to the extracted field and a fresh row variable. The extracted field is inserted into r_2, yielding r_2', and $S(r_1)$ and r_2' are then unified.

In the usual unification algorithm, there is an occurs check; here we have occurs checks for each kind of variable.

4.2.5 The implementation of recursive value bindings

In this section we will describe the implementation of recursive value bindings, i.e. the implementation of rule 27, which is a little tricky.

To recall, a recursive value binding is of the form `rec` *valbind*, and a *valbind* is of the form *pat*=*exp*⟨`and` *valbind*′⟩. We collect all the variables which will be bound; this can easily be done by traversing the left-hand side pattern *pat*. Each of these variables is then bound to a fresh type variable, yielding a variable environment VE, and the *valbind* is then elaborated as usual (as a non-recursive value binding) in a context extended with this VE — to allow for recursion. The elaboration of *valbind* yields a substitution S and a variable environment VE'. We then apply the substitution to the variable environment VE yielding $S(VE)$. The *valbind* is then traversed, and for every variable v, which is to be bound and hence occurs in a *pat*, the types bound to v in $S(VE)$ respectively VE' are unified. If the unification fails, error information is inserted into the info field of the atomic pattern containing v as its leaf and a bogus substitution (the identity substitution) is returned. If unification succeeds, the resulting substitution is returned. The insertion of error information is the reason for the explicit traversal of the *valbind*, otherwise we could simply have used the earlier collected set of variables. Name the resulting substitution S'. As the result of the elaboration of the recursive value binding, we return a triple containing $S' \circ S$, the variable environment $S'(VE')$ and the possibly error-annotated abstract syntax tree.

The reason this check is to be performed, is that VE in rule 27 both occurs before the turnstile and as the result of the elaboration (after $\Rightarrow$).

As an example, consider the following simple recursive value binding.

```
    val rec f = fn (x: bool) => f 2
```

The elaboration is performed this way: `f` is bound to the type variable α in VE, and then the expression is elaborated, whereby α is unified with $int \to \beta$ yielding the substitution S, and `f` is bound to the type $bool \to \beta$ in VE'. Then the type for `f` in $S(VE)$ and in VE', i.e., $int \to \beta$ and $bool \to \beta$ are unified; this unification fails and thereby it is discovered that the value binding should not be accepted.

According to the syntactic restriction, Definition Section 2.9, the *exp* of a *pat*=*exp* in a recursive value binding must be of the form `fn` *match*, i.e., of function type. Even though many pattern forms therefore automatically are in conflict with elaboration, the error traversal of *pat* is defined for all pattern forms.

4.2.6 Miscellaneous

To make it easier to study the Elaborator we give some comments on minor things of the implementation.

Infixes

As explained in Section 3.4

> The abstract syntax datatype contains the expression constructor `UNRES_INFIXexp`, the pattern constructor `UNRES_INFIXpat` and the declaration constructor `UNRES_FUNdec`. These are generated by the parser, but eliminated during the parsing post-pass, so that the abstract syntax tree passed to elaborator will never see `UNRES` syntax.

This is why the elaborator calls the function `Crash.impossible` for the `UNRES` constructor cases. The `Crash.impossible` function is used for internal errors — it takes a string, which it prints, as argument and raises an exception `CRASH`.

Auxiliary functions

In functor `ElabDec`, the elaborator proper, a number of simple auxiliary functions are used.

We have a number of functions named with suffixes "`Conv`" and "`Error`", which are simply used to insert information into the info field of the abstract syntax tree. The following four functions are used to insert type information and label information into some of the info fields: `addTypeInfo_CON`, `addTypeInfo_EXCON`, `addTypeInfo_LAB` and `addLabelInfo` — this information is used by the Compiler (Chapter 6).

The function `initialTE` is used to find the type environment to be used initially in the elaboration of a *datbind*. The idea used here is in fact the same as the one used in Appendix A of the Commentary in the principality proof, pp. 126–128. To recall, the idea is to bind the declared type constructors to type-structures of the form $(t^*, \{\})$, where t^* is a fresh type name with the correct arity and with equality attribute set to false. The empty constructor environment is filled in later (by function `elab_conbind`), and the correct equality attributes are determined later when maximising equality, [11, Section 4.9].

4.3 The Implementation of Semantic Objects and Operations

In this section we describe the implentation of the semantic objects and operations corresponding to [11, sections 4.1–4.9].

The specification of the semantic objects for the core and the operations on these objects is given by the following signatures. (The column *Specifies* is what the signature roughly specifies.)

Signature	*Specifies*
`STATOBJECT`	Types, typeschemes, typefunctions, substitutions, unification
`ENVIRONMENTS`	Environments, Closure-functions, Equality-maximization
`STATOBJECT_PROP`	Propagated semantic objects and operations
`ENVIRONMENTS_PROP`	Propagated semantic objects and operations

The `_PROP` signatures specify the semantic objects for the Core as required by systems that handle both Core and Modules, while the other two signatures specify some operations, which are only needed in the implementation of the static semantics for the

Core. For instance, functor `ElabDec` is parameterized on `STATOBJECT` and `ENVIRONMENTS`, while functor `ModuleEnvironments`, which implements some of the semantic objects and operations needed in Modules elaboration, is parameterized on `STATOBJECT_PROP` and `ENVIRONMENTS_PROP`.

Notice that the implementation of the semantic objects has been divided into two, as the implementation of environments need not know the exact implementation of the underlying semantic objects.

The files corresponding to the signatures are named after the signatures but with an extension `.sml`.

The functors `StatObject` and `Environments` implement the semantic objects. When the system is built, these functors are applied, yielding unconstrained structures, which afterwards are constrained by the above mentioned signatures such that we obtain the desired structures. This takes place in functor `Basics` in file `KitCompiler.sml`.

Most of the semantic objects and operations are implemented in a straightforward manner. Below we describe those semantic objects and operations that perhaps deserve a bit of explanation. The presentation follows the Definition Section 4.1–4.9.

4.3.1 Type Variables

The following is taken from functor `StatObject` and is our implementation of type variables.

```
datatype TyVar =
    ORDINARY of {id: int, equality: bool,
                 imperative: bool, overloaded: bool}
  | EXPLICIT of SyntaxTyVar
```

The thing to notice is the division into explicit type variables and ordinary type variables. The explicit type variables are implemented by the syntactic class of type variables (as specified by signature `TYVAR`). While ordinary type variables have attributes (equality, imperative and overloaded), this is not needed for the explicit type variables as the value of the attributes equality and imperative can be obtained from their syntactic appearance, and attribute overloaded is not needed as explained in section 4.2.4.

4.3.2 Environments

The contexts and environments are implemented by the following datatype (from functor `Environments`).

```
datatype
    Context  = CONTEXT of {T: TyNameSet, U: TyVarSet, E: Env}
and Env      = ENV of {SE: StrEnv, TE: TyEnv,
                              VE: VarEnv, EE: ExConEnv}
```

```
        and StrEnv   = STRENV of (strid, Str) FinMap.map
        and TyEnv    = TYENV of (tycon, TyStr) FinMap.map
        and TyStr    = TYSTR of {theta: TypeFcn, CE: ConEnv}
        and VarEnv   = VARENV of (id, VarEnvRangePRIVATE) FinMap.map
        and ConEnv   = CONENV of (con, TypeScheme) SortedFinMap.map
        and ExConEnv = EXCONENV of (excon, Type) FinMap.map
        and Str      = STR of {m: StrName, E: Env}
```

There are two things to notice here. First the range of the variable environment is implemented by the datatype

```
        datatype VarEnvRangePRIVATE =
            LONGVARpriv of TypeScheme
          | LONGCONpriv of TypeScheme * con list
          | LONGEXCONpriv of Type
```

This datatype is only used internally in functor `Environments`; the lookup operations return values of the type

```
        datatype VarEnvRange =
            LONGVAR   of TypeScheme
          | LONGCON   of TypeScheme
          | LONGEXCON of Type
```

The reason for the difference is that the above mentioned (in Section 4.2.6) function `addTypeInfo_CON` needs the extra information concerning fellow constructors to insert the correct information. By the way, notice the tagging of the types, needed to resolve identifiers as mentioned in Section 4.2.3.

The range of a variable environment is defined in the Definition to be a type scheme, so why is the domain of the constructor `LONGEXCON` *Type* instead of *TypeScheme*? The reason is simple: the range of an exception environment is defined in the Definition to be Type and exceptions enter the variable environment by copying from the exception environment, c.f. rule (21), applying an implicit injection (from Type to TypeScheme) to each type in the range of the exception environment. As we have already chosen to tag the range of a variable environment, we can avoid applying this injection by using *Type* instead of *TypeScheme* in the domain of `LONGEXCON`.

The other thing to notice is that the constructor environment is implemented using a sorted finite map, while the other environments are implemented using (unsorted) finite maps. The reason is simply that is makes it a lot easier to implement the operation `equalCE` which is needed in the Modules Elaborator (to check for enrichment, c.f., Definition Section 5.10.)

In Section 7.5 the implementation of the pervasives (given in Figure 23 in Appendix A of the Definition) is described. A small number of the pervasives are inserted manually into an initial environment. This naturally takes place in `Environments`. Everything is as you would expect, except that we assign `ref` an imperative type when used as a constructor in a pattern. This is in accordance with the Definition (cf. rule 43 and p. 28 line 12–13 which says that $C(con)$ is taken to stand for $(VE \text{ of } C)(con)$, and VE0 in Figure 23 where `ref` is assigned the type scheme $\forall$'_a.'_a$\rightarrow$'_a.) But it implies that the user cannot define `!` with the type given in VE0. The declaration `fun !(ref x) = x` will assign the type '_a `ref`$\rightarrow$'_a to `!`.

Moreover, note that the only reason for explicitly declaring the type `bool` is that it is used in the initial type schemes for the overloaded relational operators.

4.3.3 Scope of Explicit Type Variables

Recall from rule 17, Section 4.6 in the Definition (see also Commentary Section 4.4), that we must be able to compute the set of scoped explicit type variables for a given *valbind*. This computation is done by function `Scoped_TyVars` in `Environments`. `Scoped_TyVars` takes a syntax tree for a *valbind* and a type variable set (the parameter `tyvarset`), which is supposed to be the U set of the context in which the *valbind* is elaborated, and returns the set of scoped type variables. The set is found by finding the set of unguarded type variables and then removing the type variables that occur in `tyvarset`. The set of unguarded type variables is found by traversing the *valbind* and recursively collecting the explicit type variables. This is done by a collection of mutual recursive functions with the prefix `unguarded_` in their names.

4.3.4 Non-expansive Expressions

Given an expression *exp*, the function `isExpansiveExp` in functor `Environments` returns true iff the expression is expansive as defined in [11, Section 4.7]. The implementation follows the definition of expansiveness closely.

4.3.5 Closure

The Clos operation defined in Section 4.8 of the Definition is implemented in `Environments`, by a function `Clos` with the following specification (from signature `ENVIRONMENTS`).

```
        val Clos : Context * valbind * VarEnv -> VarEnv
```

The implementation follows the Definition closely. We will only like to point out that the type variables in the U set of the *Context* are considered as free type variables which is because they are going to be quantified later (as they are scoped further out). To find the set of type variables which are to be quantified, we use a function, `isExpansiveId`. For a given value binding `isExpansiveId` returns a function which, for every variable bound in the value binding, returns true, iff it is bound in a *pat* = *exp* where the *exp* is expansive. Moreover, when we for a type scheme in the range of *VE* have found the set of type variables which are to be quantified, the function `Close`, implemented in `StatObject`, is used to perform the actual closing of the type scheme.

4.3.6 Type Structures and Type Environments

The Definition declares [Section 4.9] that "All type structures occurring in elaborations are assumed to be well-formed." However, it is not necessary explicitly to check that this assumption holds in the elaborator. The reason is the following. Given that the syntactic constraint (Definition, section 2.9, third bullet) "Any tyvar occurring within the right side [of a *typbind* or *datbind*] must occur in *tyvarseq* [of the *typbind* resp. *datbind*]" is met, the type functions become well-formed, c.f. rule 28 and its comment. Moreover, the type structures generated during elaboration, using rule 28 and 29 are then well-formed too — this is easy to see, just compare the definition of well-formed type structure with the generated type structures in the two rules. Also, note that the unification (admissification) of modules preserves well-formedness.

The function `maximize_equality` in `Environments` implements the earlier mentioned maximisation of equality (see section 4.2.6). The implementation follows the description in the Commentary p. 52 directly, hence we do not describe it here.

4.3.7 Unification

The unification of types is implemented in `StatObject`. The implementation is straightforward, except for the unification of record types which we have described above. However, one needs to be careful to check the attributes of the type variables since, as mentioned, not all type variables can be unified — this is implemented in functions `checkAttributes` and `unifyExplicit`.

4.4 Files

The relevant files are listed below. They all reside in directory `Common`.

File	**Contains**
`ELABDEC.sml`	Signature for the Core Elaborator
`ENVIRONMENTS.sml`	Signature for the environments
`ENVIRONMENTS_PROP.sml`	Signature for the propagated environments
`STATOBJECT.sml`	Signature for semantic objects
`STATOBJECT_PROP.sml`	Signature for propagated semantic objects
`ElabDec.sml`	The Core Elaborator
`Environments.sml`	The environments of the semantic objects
`StatObject.sml`	Semantic objects
`KitCompiler.sml`	Builds the Kit using functor applications
`TYVAR.sml`	Signature for the syntactic category TyVar

Chapter 5

Modules Elaboration

5.1 Introduction

This section describes the Modules Elaborator of the ML Kit. We give an overview of the elaborator and explain the more intricate parts in more detail; in particular we describe how and where admissification is checked. We also describe the implementation of the static objects and operations used in the elaborator.

The outline of the chapter is as follows. Section 5.2 gives an overview of the Elaborator. Then we describe the handling of elaboration errors and the implementation of syntactic restrictions. Sections 5.5–5.19 describe the implementation of the semantic objects and operations. The constituent files of the Modules Elaborator are described in Section 5.20.

We assume the reader is familiar with algorithm W [8].

The order of the subsections of the present Chapter follows the order of the subsections of Section 5 of the Definition (the Static Semantics for Modules).

5.2 Elaboration

Functor `ElabTopdec`, located in the file `/Common/ElabTopdec.sml`, implements the elaborator for the Modules, i.e., it implements the static semantics for the Modules as defined in [11, Section 5]. Recall from Section 2.2.5 that functor `ElabTopdec` is parameterised on the objects of the static semantics of the Modules and that it returns a structure which can be constrained by signature `ELABTOPDEC`.

The rules defining the static semantics in the Definition are non-deterministic. Non-determinism is used to express sharing. Because of this non-determinism, the rules cannot be implemented directly. Hence we need an algorithm which resolves the non-determinism. Moreover we are interested in an algorithm with the properties that if it succeeds, it returns the principal signature and if the signature expression to be elaborated cannot elaborate according to the static semantics, then the algorithm fails. Our algorithm is based on the proof of the Principality Theorem [10, Appendix A.2]. The algorithm resembles W, where structures correspond to types, signatures correspond to type schemes, and realisations correspond to substitutions.

Recall the part of the Modules elaborator that deals with structure expressions from Section 2.2.5 — the implementation contains one function `elab_`*syn* for every phrase class *syn*.

5.3 Elaboration Errors

If an error occurs some time during elaboration of a subexpression, we insert error information into the node of the syntax tree for the subexpression, just as for the Core elaborator (see Section 4.2.1).

5.4 Syntactic Restrictions

The Definition states (see also Commentary, Appendix D) a number of syntactic restrictions [11, Section 3.5]. As mentioned earlier (Chapter 3) we have chosen to deal with these during elaboration.

The following additional comments to the rules in the Definition indicate where such checks are done.

Rule	*Comment*
62	By the syntactic restrictions $strid \notin \mathrm{Dom}SE$.
69	By the syntactic restrictions $sigid \notin \mathrm{Dom}G$.
82	By the syntactic restrictions $var \notin \mathrm{Dom}VE$.
83	By the syntactic restrictions $tycon \notin \mathrm{Dom}TE$, and *tyvarseq* must not contain the same variable twice
84	By the syntactic restrictions $tycon \notin \mathrm{Dom}TE$, $\mathrm{Dom}CE \cap \mathrm{Dom}VE = \emptyset$, *tyvarseq* must not contain the same variable twice, and any *tyvar* occuring in *condesc* must occur in *tyvarseq*
85	By the syntactic restrictions $con \notin \mathrm{Dom}CE$.
86	By the syntactic restrictions $excon \notin \mathrm{Dom}EE$.
87	By the syntactic restrictions $strid \notin \mathrm{Dom}SE$.
94	By the syntactic restrictions $funid \notin \mathrm{Dom}F$.
99	By the syntactic restrictions $funid \notin \mathrm{Dom}F$.

These comments to the rules in the Definition tell us exactly which properties to check and also where to check them.

If at some point a restriction is violated, error information is inserted into the syntax tree.

5.5 Semantic Objects

The specification of the semantic objects for the Modules and the operations on these objects are given by the following signatures.

Signature	*Specifies*
`MODULE_STATOBJECT`	Simple semantic objects and operations in the static semantics
`MODULE_ENVIRONMENTS`	Environments and operations on these

Notice that the implementation of the semantic objects has been divided into two, as the implementation of environments need not know the exact implementation of the underlying semantic objects.

The files corresponding to the signatures are named after the signatures but with an extension `.sml`.

The functors `ModuleStatObject` and `ModuleEnvironments` implement the semantic objects. They are both parameterised on the semantic objects of the Core, as specified by signatures `STATOBJECT_PROP` and `ENVIRONMENTS_PROP`.

Consider the further compound semantic objects for the Modules [11, Figure 11] and the simple operations on these [11, Section 5.1]. The implementation of these is straight out of the book; StrNameSet, NameSet, Sig, and FunSig are implemented in `ModuleStatObject`, and the rest in `ModuleEnvironments`.

The way the Modules Elaborator "inherits" the semantic objects of the Core Elaborator can be summarised by the following simplified program outline.

```
functor StatObject(...) = ...
              (* Core objects including e.g. type realisations
                 for use in Modules Elaborator *)
structure StatObject = StatObject(...);
structure CoreStatObject : STATOBJECT = StatObject
              (* Core Elaborator's view of Core Objects *)
structure StatObjectProp : STATOBJECT_PROP = StatObject
              (* Modules Elaborator's view of Core Objects *)
functor ModuleStatObject(O : STATOBJECT_PROP) :
       MODULE_STATOBJECT =
              (* bases, signatures, functor signatures etc. *)
structure ModuleStatObject = ModuleStatObject(StatObjectProp)
```

We must be able to deal with assemblies of semantic objects as all semantic objects occuring in the inference tree for a signature expression are required to be admissible ([11, Section 5.5]). In the Definition, assemblies are referred to simly as a collection of semantic objects. Assemblies are, however, only used during admissification [10, Section 10.4] and to check for cover. Therefore, we can make do with the following types for assemblies, defined in functor `ModuleEnvironments`, which only represents the "essential" parts of an assembly of semantic objects.

```
type offspring_Str = (strid, StrName) FinMap.map
 and offspring_Ty  = (tycon, TyStr  ) FinMap.map
```

```
type offspring     = {structures : offspring_Str,
                      types : offspring_Ty};
type Assembly_Str = (StrName, offspring)FinMap.map
 and Assembly_Ty  = (TypeFcn, TyStr)FinMap.map
type Assembly     = {m_arcs : Assembly_Str,
                     theta_arcs : Assembly_Ty}
```

Notice that the representation is "flat", in the sense that `offspring_Str` has `StrName` as its range type (instead of `offspring`). Consider an assembly consisting of the structure

```
structure S0 =
  struct
    structure S10 =
      struct
        structure S20 = struct end
      end
    structure S11 =
      struct
      end
  end
```

The representation of the assembly is then

```
{ m_arcs = { m0  ↦ { structures = { S10 ↦ m10,
                                    S11 ↦ m11 },
                     types =      {} },
             m10 ↦ { structures = { S20 ↦ m20 },
                     types =      {} },
             m11 ↦ { structures = {},
                     types =      {} },
             m20 ↦ { structures = {},
                     types =      {} },
  theta_args = {} }
```

where we as usual use `{}` around records and {} around finite maps. This flat representation is chosen because it makes the implementation of admissification easier.

Also notice that for a type structure θ in the domain of `theta_arcs`, θ is mapped to (θ, CE) for some CE. That is, θ is represented twice. (Actually, there is no good reason for this.)

Functor `ModuleEnvironments` implements operations to build assemblies from other semantic objects (we must be able to turn a basis into an assembly when we start elaborating a signature expression, see Section 5.18 below.) The functor also implements functions satisfying the following specifications (from signature `MODULE_ENVIRONMENTS`)

```
val Alookup_Str         : Assembly * StrName -> offspring_Str
and Aoffspring_Str_Fold : (((strid * StrName) * 'b) -> 'b) -> 'b ->
                           offspring_Str -> 'b
val Alookup_Ty          : Assembly *  StrName -> offspring_Ty
and Aoffspring_Ty_Fold  : (((tycon * TyStr) * 'b) -> 'b) -> 'b ->
                          offspring_Ty -> 'b
val Alookup_TypeFcn     : Assembly * TypeFcn -> TyStr
```

The `_Fold` functions are simply functions which fold a function over the elements in the range of a finite map, and the `Alookup_` functions are, of course, lookup functions. Other operations on assemblies are also implemented; some of them are described in the relevant sections below.

5.6 Consistency

As explained in the Commentary (Section 10.5 and the solution to Exercise 10.4, p. 145) consistency need only be checked during admissification. Admissification is implemented by functor `ModuleUnify`; the implementation of the necessary check for consistency of type structures [11, Section 5.2], function `CheckCEdom`, is in this functor.

5.7 Well-formedness

To recall ([11, Section 5.3]), an assembly A is well-formed if every type environment, signature and functor signature occuring in A is well-formed.

Functor signatures are not derivable solely by rules $\mathcal{R}_{\mathrm{sig}} \cup \{\text{rule } 65\}$ (see Chapter 10 of the Commentary where $\mathcal{R}_{\mathrm{sig}}$ is defined); the other rules that are needed in a derivation of a functor signature preserve admissibility and we need not check for well-formedness of functor signatures (Commentary Section 10.5).

It suffices to check for well-formedness of signatures in rule 65, see page 92 and Section 11.4 of the Commentary. The implementation of this check is described in Section 5.18 below.

Well-formedness of type structures [11, Section 4.9] need only be checked during admissification (Commentary Section 10.5 and solution to Exercise 10.4). The implementation of the check occurs in functor `ModuleUnify` (function `CheckTyStrWF`).

5.8 Cycle-freedom

It suffices to check for cycle-freedom during admissification (Commentary Section 10.5). Function `cyclic` in functor `ModuleEnvironments` takes an assembly as argument and returns true iff the assembly is cyclic. The cycle test is performed by a mindless depth-first search (for each structure name).

5.9 Admissiblity

Admissibility, [11, Section 5.5], is implemented by separate checks for consistency, well-formedness and cycle-freedom as described above. (No admissibility predicate is provided or needed.)

5.10 Type Realisation

Type realisations are implemented by functor `ModuleStatObject`. Actually, the operations are provided by one of the argument structures to `ModuleStatObject`. The argument structure matches signature `STATOBJECT_PROP`, which in turn specifies part of the static objects and operations for the Core. Type realisations are only needed in the static semantics for the Modules, but the implementation of type realisation needs to know the representation of type names and type functions. Therefor, type realisations are implemented in functor `StatObject`.

5.11 Realisation

A realisation, [11, Section 5.7], is implemented in functor `ModuleStatObject` simply as a record consisting of a type realisation (see above) and a structure realisation (implemented by a SML function). Operations for applying realisations to the representations of semantic objects are also implemented here.

5.12 Type Explication

Check for type explication [11, Section 5.8] is implemented by a function `type_explicit`, specified in signature `MODULE_STATOBJECT` by

```
        val type_explicit : NameSet * Str -> bool
```

and implemented in functor `ModuleStatObject`. The nameset argument is supposed to be the set of rigid names of the basis in which the check for type explicitness is made. Notice that type explication in the Definition, [11, Section 5.8], is defined for a signature (N)S; when the above function is called with arguments `NofB` and `S`, the set of bound names N mentioned in the Definition is equal to those free names of `S` that are not in `NofB`.

The function simply collects the set of explicit type names in the structure argument and compares this with set with the set of bound names, and returns true iff they are equal.

5.13 Signature Instantiation

Signature instantiation [11, Section 5.9] is implemented by function `instanceSig` in functor `ModuleStatObject`; given a signature (N)S it simply returns a structure where the names bound by N have been instantiated to fresh names (by applying a realisation).

5.14 Functor Signature Instantiation

We have not implemented a function directly corresponding to functor signature instantiation [11, Section 5.10]. Instead a function, specified by

```
datatype 'a MatchResult =  OK of 'a
                         | ERROR of ErrorInfo

val funsigMatchStr : FunSig * Str -> Sig MatchResult
```

in signature `MODULE_STATOBJECT`, is provided. Given a functor signature and a structure, it returns the signature of the actual result of the functor application, if the structure matches the argument signature of the functor signature; otherwise error information is returned. The function is implemented using implementations of enrichment and signature matching described below.

5.15 Enrichment

Enrichment is implemented by a function `enrichesS` in functor `ModuleStatObject`, straight out of the Definition, Section 5.11. As enrichment is only needed to implement signature matching, which is also implemented in functor `ModuleStatObject`, it is not specified in signature `MODULE_STATOBJECT`; thus the function is therefore not visible outside the functor.

5.16 Signature Matching

Signature matching [11, Section 5.12] is defined by a combination of instantiation and enrichment. But, of course, we cannot just take an arbitrary instance of the signature and then check whether or not that instance enriches the structure. We must instantiate the signature via a realisation that maps the flexible names of the signature to corresponding names in the structure to be matched. Therefore signature matching proceeds by a recursive traversal of the signature and structure, instantiating flexible names along the way; afterwards enrichment is checked. Signature matching is implemented by function `sigMatchStr`, the recursive instantiation is performed by function `sigMatchRea`; both functions are implemented in functor `ModuleStatObject`.

When a structure is matched against a signature, the signature has been accepted earlier, and the signature is therefore type-explicit. Hence all flexible type names occur in the type environments of the signature (recall the definition of type-explicitness [11, Section 5.8]). This is exploited in `sigMatchRea` — it is enough to recursively traverse structure and type environments. A type name is flexible if it occurs as a type function in a type structure in the range of a type environment *and* also occurs in the set of flexible type names of the signature.

5.17 Principal Signatures

5.17.1 Cover

Cover, [11, Section 5.13], is needed to implement rule 65 (see Section 5.18 below) as explained in the Commentary (Chapter 11). Function `covers`, implemented in functor `ModuleEnvironments`, takes a name set, an assembly, and a structure and checks whether or not the assembly covers the structure on the name set. Notice that cover in the Definition is not defined in terms of an assembly but in terms of a basis; as seen from the definition of cover, however, we only need the “skeleton” of the basis and the set of rigid names of the basis to check for cover; hence the above mentioned types of the arguments to `covers`.

The implementation is simple; for every substructure (m, E) in the given structure, if m is rigid (occurs in the given name set), the offsprings recorded in the assembly for m are looked up. These offsprings correspond to the type and structure environments for the rigid structure in the basis. Then it is checked that the domains of the offsprings are supersets of the domains of the structure and type environments of E. If this is not the case, the assembly does not cover the structure.

5.17.2 Equality-Principal Signature

Equality principality is implemented in functor `ModuleStatObject` by function `equality_principal`, which takes a structure, S, and a name set, N, as arguments. The name set N is supposed to be the set of rigid names in the basis, so that the flexible names of S are the free names of S less the names in N. The implementation follows the Definition [11, Section 5.13] closely (see also [10, page 51]). First the set of type names T_0 is found, and by fixed point iteration the maximal set of type names which can be made to admit equality is computed, resulting in a subset of T_0. This computation makes use of function `maximize_equality` implementing maximisation of equality for type environments, c.f., Section 4.2.6. When a fixed point is reached, it is checked that the signature respects equality.

5.18 Inference Rules

As mentioned in the Section 5.2, we implement the rules that define the static semantics of the modules [11, Section 5.14], by an algorithm that resembles algorithm W. Almost all rules are implemented straightforwardly; in this section we therefore only describe the fun rules! First, however, we summarise where assemblies are used.

5.18.1 Assemblies

Recall that all semantic objects occurring in the signature expression are required to be admissible and that assemblies are needed during admissification.

The following functions take an assembly as argument besides the other usual arguments (the expression to be elaborated and the semantic objects against which the expression is to be elaborated).

Function	*Corresponding rule(s)*
`elab_sigexp'`	63–64
`elab_spec`	70–81
`elab_strdesc`	87
`elab_shareq`	88-90

The reason it is exactly these functions that take an assembly as argument is as follows: the assembly is needed during admissification, which is invoked for each sharing equation in the original signature expresion; but sharing equations occur in specifications which occur in signature expressions and signature expressions occur in structure descriptions. The elaboration of any such phrase can be affected by sharing constraints it contains; that is the reason why the four functions above must take an assembly as a parameter.

The following functions return an assembly besides the usual return values (the semantic object obtained by elaboration and the modified syntax tree.)

Function	*Corresponding rule(s)*
`elab_sigexp'`	63–64
`elab_spec`	70–81
`elab_typdesc`	83
`elab_datdesc`	84
`elab_strdesc`	87

These are the functions that generate semantic objects needed during admissification; for instance, type and datatype descriptions elaborate to type structures and we must check for well-formedness of type structures during admissification. The functions `elab_sigexp'` and `elab_spec` return an assembly in order to propagate the generated semantic objects from substructure expressions.

5.18.2 Rule 65 — Principal signatures

Function `elab_sigexp` takes a basis and a signature expression and elaborates the signature expression in that basis. We first turn the basis into an assembly and then proceed to elaborate the signature expression (corresponding to an application of either rule 63 or rule 64). Then we simply follow the description in the Commentary p. 113, and check for cover, well-formedness of the signature, type-explicitness, and return the equality principal signature if it exists. The only operation which we have not already described is the check for well-formedness of a signature. It is implemented in functor `ModuleStatObject` by function `wellformedsig` specified in `MODULE_STATOBJECT` by

```
        val wellformedsig : NameSet * Str -> bool
```

The nameset argument is supposed to be the set of rigid names in the basis in which the check is performed. The implementation recursively traverses the structure argument; for each substructrue (m, E) it is checked that if m is rigid (is in the nameset), then all the free names in E are rigid too. If this holds for all the substructures then the signature is well-formed.

5.18.3 Rule 84 — Datatype descriptions

Here we use the same idea as for datatype bindings in the Core Elaborator, see Section 4.2.6.

5.18.4 Rules 88–90 — Sharing equations

Recall from [10, Section 10.4] that the admissification process must be invoked for each sharing specification. Admissfication is implemented in functor `ModuleUnify` by the functions `unifyTy` and `unifyStr`, which are specified in signature `MODULE_UNIFY`:

```
        datatype UnifyResult =
            OK of Realisation
          | ERROR of ErrorInfo
        val unifyStr: NameSet * Assembly *
                      (StrName * longstrid) list -> UnifyResult
        and unifyTy : NameSet * Assembly *
                      (TyStr   * longtycon) list -> UnifyResult
```

If the admissification succeeds, it returns the most general admissifier (c.f., Theorem 10.3 in the Commentary), and this realisation is returned as part of the result of elaborating the sharing equation (just like the functions in the Core Elaborator return a substitution.)

For a structure sharing specification we look up the structures bound to the structure identifiers in the sharing specification, and call `unifyStr` with a list of the names of the structures. Actually, as one sees from the type of `unifyStr`, each structure name is paired with the structure identifier via which it was found — this is for error reporting only.

For a type sharing specification we similarly look up the type structures bound to the type constructors and call `unifyTy`.

The implementation of `unifyTy` and `unifyStr` follows the proof of Theorem 10.3 in the Commentary and the notes given in the solution to Exercise 10.4 in the Commentary. As the code is well-commented and follows the description in the Commentary very closely, we will give only one little remark here. In `unifyTy` we replace the constructor environments in the argument type structures by constructor environments looked up in the assembly. The reason can be seen from the following example. Consider the elaboration of the second sharing specification in the following signature expression

```
sig
  datatype t1 = C of int
  type t2
  sharing type t1 = t2
  datatype t3 = D of int
  sharing type t2 = t3
end
```

If we simply look up the constructor environment for `t2` in the basis we get an empty constructor environment, ignoring that `t2` shares with `t1` and the second sharing would (wrongly) appear to be legal. But since the Kit looks the type structures up in the assembly we get the non-empty constructor environment (with domain {`C`}) and so the Kit spots that the sharing specification violates the third condition of consistency (conflicting non-empty constructor environment domains — Definition, Section 5.2).

5.19 Functor Signature Matching

We have not implemented functor signature matching [11, Section 5.15] as this is not required for the execution of programs.

5.20 Files

The relevant files are listed below. They all reside in directory `Common`.

File	**Contains**
`MODULE_STATOBJECT.sml`	Signature for the semantic objects
`MODULE_ENVIRONMENTS.sml`	Signature for the environments
`MODULE_UNIFY.sml`	Signature for the admissication process
`ModuleStatObject.sml`	Semantic objects
`ModuleEnvironments.sml`	The environments of the semantic objects
`ModuleUnify.sml`	The admissification process
`ElabTopdec.sml`	The Modules Elaborator corresponding to the inference rules

Chapter 6

Compilation from the Core to Lambda

6.1 Introduction

This chapter describes a compiler from the SML core language to a simple intermediate code based on the lambda calculus. It forms an optional part of the SML Kit, and the interface presented to the rest of the Kit is the same as that presented by the "pure" interpreter. The lambda compiler is not intended to be an example of how to generate highly efficient code for SML; instead, it serves to illustrate in a clear, modular fashion some of the techniques which are needed to compile core SML into a simple intermediate language.

The Kit is, inasmuch as it is possible, a direct implementation in SML of the SML Definition [11], such that the internal structure of the compiler reflects the objects and inference rules of the formal semantics. It contains a typechecker which is a fairly direct coding of the static semantics of SML and a "pure" interpreter which is a corresponding coding of the dynamic semantics. The purpose of this paper is to describe a modular extension to the Kit: a replacement interpreting phase which actually compiles SML to a simple intermediate code and executes this. This extension, which we shall refer to as "the compiler", has the same external interface as the interpreter of the dynamic semantics (henceforth "the interpreter"), all differences being internal. (In fact, there is one difference: the compiler will report inexhaustive and redundant matches, since it has the machinery to do so; the interpreter does not.)

Our motivation in designing and implementing the compiler was not efficiency; instead, it was to produce a system to illustrate the techniques needed to compile SML into a simple intermediate language. If these techniques are implemented in a clear and modular fashion, gaining efficiency is largely a matter of refinement. However, a compiler which has been constructed with efficiency as a driving force in its design is unlikely to demonstrate its internal structures and algorithms in a clear and modular fashion.

A motivation for the SML Kit was to have a system which was complete in its implementation of the SML language and yet open enough in terms of its internal structure and interfaces to allow aspects of its operation to be investigated, altered and experimented with. The desire was for a tool which would allow changes or additions to the formal semantics to be implemented with the minimum of effort, so that they could actually be seen in operation and played with (and this aim has been achieved in a number of projects, including Extended ML [14]). While there are a number of SML compilers in existence or under development at present, none can be considered simple enough to allow this kind of experimentation. In addition, large, commercial-quality compilers tend to contain extra machinery required for efficient code generation, and this tends to obscure modularity. Also, there is no motivation on the part of compiler writers to produce a piece of software which reflects the Definition in its internal structure—all that is important for such an implementation is adherence to the language semantics "on the outside" (and even here, compilers can vary).

Is this then perhaps a good reason *not* to implement any kind of compiler in the SML Kit? Possibly; but it was considered a useful exercise for the reasons outlined below, and in addition, the highly modular nature of the Kit allowed the compiler to be integrated in isolation with no compromises to the structure or operation of the rest of the system, apart from some minimal information required from the typechecker. The Kit can be built as an interpreter or as a compiler from a large body of shared code—30000 lines of the system (including the generated parser and all the static semantics machinery) is common between the interpreter version (an additional 1500 lines) and the compiler version (an additional 5500 lines).

The Kit provides a good testbed for experimentation at the level of the formal semantics. New ideas for typechecking, for example, would be formulated in the static semantics and could them be implemented in a fairly direct manner in the Kit. Similarly, alterations in the semantics of program execution could be described formally and then implemented. However, there are experiments and projects which would benefit from a simple, modular, implementation of the complete SML language and yet might not fall into an area which can be addressed by the formal semantics. This is probably unlikely in the case of the static semantics—type systems are, by their very nature, formal, and can only be dealt with in a satisfactory manner using semantic techniques. In the case of the dynamic semantics, the situation is rather different. Pragmatics still plays a major part in compiler research, and program execution can be considered independently of the dynamic semantics. In particular, the provision of a functional intermediate language (in this case, one related to the lambda calculus) means that experiments which are often performed on the source language (such as code instrumentation) might be done at the level of the cleaner, simpler, intermediate language instead.

For such pragmatic experiments, it was felt that the dynamic semantics of the Definition fell at too high a level to be of direct use. While the dynamic semantics of SML is fairly simple, it is simple at a level which might fail to convince a compiler writer. The dynamic semantic objects are at quite a high-level, are built from high-level abstractions like finite maps, and sometimes have fairly complex behaviour. By contrast, research in compilation techniques is often concerned with representations of data objects and the ways in which such objects can be efficiently manipulated. The notion of representation is different to a compiler writer with his concept of machine words and memory, to that of a theorist who formulates rules in terms of environments and maps, and concepts of

```
signature EVALTOPDEC =
  sig
    type DynamicBasis
    type topdec

    datatype result = SUCCESS of DynamicBasis
                    | FAILURE of string

    val eval: DynamicBasis × topdec → result
  end
```

Figure 6.1: The signature EVALTOPDEC

efficiency (whether at the level of algorithm complexity or numbers of operations) do not arise at all at the semantic level.

A greater impetus is given by the fact that a large portion of the SML Kit can be shared between the interpreter and the compiler, and the compiler can benefit from the parser and typechecker for the entire SML language, plus the overall modular skeleton of the interpreted system.

It should be stated that some of the main ideas in the compiler (such as the adoption of an intermediate lambda language with unique variables, and the use of higher-order functions to compile declarations) were first seen in Standard ML of New Jersey [3], and the reader is referred to [2] for a more extensive coverage of SML code generation, together with some of the techniques employed in the Kit. Suffice to say that the Kit Compiler is a lot simpler in overall structure and function, but operates in a similar way to the back-end and runtime system of SML/NJ.

6.2 Structural Overview

The SML Kit's evaluation stage revolves around the concept of *dynamic basis* presented in the Definition. The evaluator takes a dynamic basis representing the current top-level environment, and a top-level declaration, and attempts to evaluate it. The evaluation is either successful, yielding the dynamic basis corresponding to the declaration, or else it fails due to an exception propagating to the top level. The `EVALTOPDEC` signature is given in Figure 7.1[1].

The interpreter and the compiler each form a subsystem which conforms to this interface. The implementation of the dynamic basis is different in each case, although both implementations provide the operations of the dynamic semantics (`E_in_B`, `E_of_B`,

[1]The code examples have been beautified slightly: "`*`" (for type tuples) is written "×", `->` (for the function type) is "→", and type variables are written "α", "β" and so on (or "$\alpha_=$", "$\beta_=$" if they have the equality attribute).

```
signature DYNAMIC_BASIS =
  sig
    type Basis

    val emptyBasis: Basis
    val initialBasis: Basis
    val B_plus_B: Basis × Basis → Basis
  end
```

Figure 6.2: The signature `DYNAMIC_BASIS`

`B_plus_B`). The rest of the SML kit accesses the evaluation modules through the interfaces provided by the signatures `EVALTOPDEC` and `DYNAMIC_BASIS` (Figure 6.2), and has no knowledge of any of their internals.

The compiler contains a number of modules associated with analysing, compiling and executing declarations. These are brought together in a single linkage module which compiles and evaluates top-level declarations. This top-level module of the compiler translates (core-level) declarations to lambda code which is fed through a simple optimiser and then executed. If successful, the execution results in a new set of dynamic bindings to add to the top-level environment. As far as the rest of the Kit is concerned, the interface is just an evaluator for declarations; the compiler is totally hidden. (At present there is some temporary clutter in this module since we do not yet have support for the modules language, and yet require support for top-level `local` declarations which can only be parsed as module-level constructs.)

The compilation phase is fairly straightforward, and we describe it in Section 6.6. The part of the compiler responsible for compiling pattern matches, however, is sophisticated and quite complex; an overview is given in Section 6.7. The interface between the two is a data structure called a *decision tree*. The compiler works at the level of dynamic objects which bear some similarity to the objects of the Definition's dynamic semantics, but are at a lower level and much simpler. The intermediate lambda language is fairly simple, as is its interpretation. The runtime objects are discussed in Section 6.4, the lambda language in Section 6.5, and the lambda interpreter in Section 6.10.

6.3 Variables and Environments

Where the dynamic semantics retains the notion of a *variable* in the source language, and builds environments as mappings from such variables, we introduce a new type of identifier, the *lvar*, which is known only to the compiler and lambda interpreter. Lvars are represented numerically and are generated uniquely during compilation. Therefore, even though the same SML variable can occur in several places in the source text, with different meanings, an lvar is bound exactly once. This property simplifies much of the compiler since a lot of environment analysis is no longer necessary; many operations on the intermediate code can be performed with respect to an lvar without worrying about the same lvar being rebound in a nested scope[2].

Corresponding to the various environments of the dynamic semantics, the compiler has two kinds, whose interfaces are shown in Figures 6.3 and 6.4. The *dynamic environment*, or DEnv, maps lvars to runtime objects. The *compiler environment*, or CEnv, maps SML variables to lvars. CEnvs are used during compilation to deal with SML identifiers occurring in the source text being compiled; DEnvs are used during execution to deal with the evaluation of lvars.

When a declaration is compiled, each variable bound by the declaration is associated with a new, unique, lvar. This mapping of SML variables to lvars constitutes the CEnv for the declaration. The code is then executed to yield a set of objects, one for each variable. These objects are bound to the lvars to form the DEnv for this declaration. Finally, the CEnv and DEnv are incorporated into the top-level environment. In subsequent declarations, the compiler will translate occurrences of the newly-bound SML variables into their corresponding lvars via the CEnv; during execution, the lvars will be mapped to their runtime values by the DEnv.

Recall from Section 6.2 that the compiler must match the signature `EVALTOPDEC` in order to link with the rest of the system. This it does by implementing the type `DynamicBasis` as a pair (`CEnv` $\times$ `DEnv`). This allows SML identifiers to be mapped to runtime objects by looking them up in the CEnv and then looking up the resulting lvars in the DEnv.

6.4 Runtime Data Objects and Representations

The Definition has a large selection of dynamic, or "runtime", data objects, including constructors, records, exception packets, and various kinds of environment. These semantic objects are defined abstractly at a fairly high-level, and some (records and environments) are defined using other high-level constructs (finite maps). The compiler has a small number of simple data objects and a single notion of environment, in which the semantic objects are implemented. This implementation requires the compiler to make choices of representation; some of the choices require information from the typechecker.

The compiler takes SML declarations and generates lambda code which manipulates these objects; the compiler therefore needs to be aware of the mapping between semantic objects and their representations. Execution of lambda code takes place in (a representation of) the top-level environment, and produces new top-level bindings in terms of compiler objects; the module responsible for the top-level environment must manipulate the objects and dynamic environments consistently, without invalidating the abstract view of program execution which the rest of the SML Kit relies upon.

The compiler's `OBJECTS` signature is shown in Figure 6.5. This is essentially an interface to a (hidden) datatype, but there are advantages to making the interface functional—in particular, closures are built in a clean way. The types `lvar` and `DEnv`, as described above, are needed for building closures, as is the type `LambdaExp` of lambda expressions

[2]and some cannot; beta-reduction of functions, for example, would require variable renaming.

```
signature DYNAMIC_ENV =
  sig
    type DEnv
    type lvar
    type object

    val emptyDEnv: DEnv
    val declare: (lvar × object × DEnv) → DEnv
    val plus: DEnv × DEnv → DEnv
    val REC: DEnv → DEnv      (* Semantics V4, p48 *)
    val lookup: DEnv → lvar → object
  end
```

Figure 6.3: The signature DYNAMIC_ENV

(Section 6.5).

The simplest runtime objects correspond to the *special values* of the Definition: integers, reals and strings. In addition, there is a *void* object used in the representation of `unit` and of nullary exception and data constructors (Section 6.6). `vector` and `select` allow vectors (non-assignable arrays) of objects to be built and decomposed. `Ref` and `deRef` build and decompose references. `equal` provides equality on objects; it respects the identity of references, and is undefined on closures.

The representation of closures corresponds very closely to that in the Definition, but has an extra component. `bodyEnv` and `recEnv` correspond to the elements E and VE in the closures of the semantics. In place of the *match*, we have an lvar and a lambda expression. Where the Definition applies the match to an argument, we instead evaluate the lambda code in an environment which binds the lvar to the argument. The dynamic environment signature (Figure 6.3) provides the *Rec* operation described in the Definition. As an aside, the types of objects and dynamic environments are mutually recursive: objects include closures which contain dynamic environments, and dynamic environments map lvars to objects.

There are a number of invariants concerning the operations on objects: the unpacking operations must be applied to objects of the correct type, `select` must use an index within an appropriate range for the length of the vector, and so on.

6.5 The Lambda Language

We now describe the lambda language generated by the compiler. The signature specifying the lambda language datatype is shown in Figure 6.6. The language is at a level usually associated with compiler intermediate codes, but the Kit interprets it directly rather than compiling it down into lower-level code. The lambda language is essentially lambda calculus with one or two simple extensions. It contains constants of the base types

```
signature COMPILER_ENV =
  sig
    type CEnv
    type var                    (* Unqualified variables *)
    type longvar                (* Qualified vars (for lookup) *)
    type excon                  (* Unqualified exception constructors *)
    type longexcon              (* Qualified excons (for lookup) *)
    type lvar                   (* Unique lambda identifiers *)

    val emptyCEnv: CEnv
    val initialCEnv: CEnv

    val declareVar: (var × lvar × CEnv) → CEnv
    val declareExcon: (excon × lvar × CEnv) → CEnv

    val plus: CEnv × CEnv → CEnv

    datatype result = LVAR of lvar
                    | PRIM      (* Looking up a value identifier will *)
                                (* either get you a real lvar, *)
                                (* or PRIM (which means you have to *)
                                (* extract the integer argument and *)
                                (* use it in a PRIM_APP lambda-exp) *)

    val lookupLongvar: CEnv → longvar → result
                                (* Only prim should give you back *)
                                (* PRIM (and even then it can be *)
                                (* overwritten) *)

    val lookupLongexcon: CEnv → longexcon → lvar
                                (* Looking up an excon must give *)
                                (* you an lvar *)

    val lvarsOfCEnv: CEnv → lvar list
  end
```

Figure 6.4: The signature COMPILER_ENV

```
signature OBJECTS =
  sig
    type lvar
    type LambdaExp              (* For building closures.   *)
    type DEnv                   (* lvar→object environment.   *)
    type object

    val void: object

    val integer: int → object
    val deInteger: object → int

    val real: real → object
    val deReal: object → real

    val string: string → object
    val deString: object → string

    val closure:
      {arg: lvar, body: LambdaExp, bodyEnv: DEnv} → object
                                (* The recEnv part is created empty.   *)

    val deClosure:
      object → {arg: lvar, body: LambdaExp, bodyEnv: DEnv, recEnv: DEnv}

    val vector: object list → object
    val select: int × object → object

    val Ref: object → object
    val deRef: object → object

    val equal: object × object → bool
  end
```

Figure 6.5: The signature OBJECTS

associated with SML's special values, as well as *switches* (simple case expressions) over them. We consider the lambda language to be *untyped*, in that there is no provision for defining or expressing types in the language. The integers, strings and reals of the lambda language can be considered to have SML types `int`, `string` and `real` respectively, but there is no notion of type abstraction, datatype and so on.

The variables of the language are the *lvars* introduced in Section 6.3. Lvars are bound in function (`FN`) expressions and recursive declarations (`FIX`) and referenced by `VAR` leaf nodes. The Definition's *special constants* map to lambda-expressions of the kind `INTEGER`, `STRING` or `REAL`, which have corresponding kinds of *object* (Section 6.4). Similarly, there is a `VOID` lambda constructor which yields a `void` object. (The name `VOID` is used rather than `UNIT` since there are many circumstances in which a `VOID` lambda term might be used, and the word `UNIT` might misleadingly suggest some relation to SML's `unit` data object.)

Function (`fn`) expressions are compiled into `FN` expressions in the lambda language. These only take a single lvar as argument, so a fair amount of work is involved in transforming SML function expressions containing matches into lambda functions; most of the work is done by the *match compiler* (Section 6.7).

Mutually recursive function definitions compile into `FIX` expressions which bind a set of lvars in parallel. (Recall that the compiler associates a new lvar with each variable in an SML declaration.) The right hand sides of the bindings are expected to be function (`FN`) expressions. The `APP` constructor represents application of one lambda-expression to another in the obvious way. `PRIM_APP` is used for pervasive functions (the *basic values* of the Definition) in a manner described in Section 6.8.

Corresponding to the many data objects of the Definition (records, data constructors, exception constructors, references), the lambda language has two distinct data manipulating constructors. `VECTOR` takes a list of lambda expressions and returns a fixed-length list of objects from which elements can be extracted using the `SELECT` constructor. (In an implementation, one would expect this to be an array with constant access time, but this is not a requirement.) `SELECT` takes an index i between 0 and $n-1$ where n is the length of the vector and returns the i'th element (and is undefined for values of $i < 0$ or $\geq n$). `REF` is a constructor for creating a reference to an object using a new address, as in the Definition.

The lambda language has conditional expressions over the three basic types (integers, strings, reals); comparison of reals is assumed to make sense because it is required by the Definition. There are three "branch" operations (`SWITCH_I`, `SWITCH_S` and `SWITCH_R`), one for each corresponding object type, although the switches have identical structure (and in fact the switch datatype is polymorphic). A switch can be considered a degenerate `case` expression: it has an *argument*, which is a lambda expression which must evaluate to a basic value (constant) of the correct type, and contains a finite map from constants to lambda expressions. If the argument is in the domain of the map the corresponding lambda expression from the range is evaluated as the value of the switch. If the map does not contain the argument, the switch evaluates and returns its optional *wildcard*. The value of an unmatched switch with no wildcard is undefined. The wildcard is optional for integer switches, since the compiler often knows that the switch is exhaustive for the possible range of values of the argument. The wildcard is mandatory, however, for string and real switches because they are generated only as a result of matches in the source SML program and can never be exhaustive. In these case the compiler always generates a wildcard which is appropriate.

Finally, the lambda language provides for exception raising and handling. `RAISE(`e`)` raises the exception packet denoted by e (and whose format is described in Section 6.6). `HANDLE(`e_1`, `e_2`)` evaluates e_1; if an exception is propagated, e_2 is evaluated to yield a `FN` expression, and the result of the `HANDLE` expression is e_2 applied to the exception packet. Any exception resulting from this application is propagated outward.

It can be seen that the interpreter for the lambda language carries an implicit dynamic exception context during evaluation; the interpreter is described in detail in Section 6.10.

The `LAMBDA_EXP` signature contains some convenient shorthand functions for building the common forms of lambda construct. Of particular note is the form

`Let(`v`, `e_1`, `e_2`)`

which evaluates to

`APP(FN(`v`, `e_2`), `e_1`)`

This is a translation which can be performed safely in the untyped lambda language; the same is not true in SML itself, due to the way in which `let`-bound variables have their types generalised.

6.6 The Declaration Compiler

6.6.1 Conventions and Representations

The declaration compiler has two purposes:

- to translate declarations in the SML core language into expressions in the lambda language;
- to generate a compiler environment (CEnv) for the variables bound in each declaration.

However, since (SML) declarations yield environments and (lambda) expressions yield values, there is no obvious translation from declarations to lambda expressions. The following, rather elegant, convention is due to Andrew Appel [3].

The compiler is a function `compileDec` with type

`CEnv` $\to$ `dec` $\to$ `(CEnv` $\times$ `(LambdaExp` $\to$ `LambdaExp))`

Every SML declaration is treated as if it had a *scope*. For top-level declarations, the scope does not actually exist, although each such declaration *dec* is treated as if it occured as part of an expression of the form

`let` *dec* `in` *scope* `end`

```
signature LAMBDA_EXP =
  sig
    type lvar
    type (α=, β) map

    datatype α option = NONE | SOME of α

    datatype LambdaExp =
        VAR of lvar           (* Lambda variables *)
      | INTEGER of int        (* Constants... *)
      | STRING of string
      | REAL of real
      | FN of lvar × LambdaExp (* Function-terms *)
      | FIX of lvar list × LambdaExp list × LambdaExp
                              (* Mutually recursive fns *)
      | APP of LambdaExp × LambdaExp (* Function application *)
      | PRIM_APP of int × LambdaExp (* Primitive function application *)
      | VECTOR of LambdaExp list (* Records/tuples *)
      | SELECT of int × LambdaExp (* Con/record indexing *)
      | SWITCH_I of int Switch (* Switch on integers *)
      | SWITCH_S of string Switch (* Switch on strings *)
      | SWITCH_R of real Switch (* Switch on reals *)
      | RAISE of LambdaExp    (* Raise exception *)
      | HANDLE of LambdaExp × LambdaExp (* Exception handling *)
      | REF of LambdaExp      (* ref(expr) *)
      | VOID                  (* nil, () etc *)

    and α Switch = SWITCH of {arg: LambdaExp,
                              selections: (α, LambdaExp) map,
                              wildcard: LambdaExp option
                              (* Wildcard mandatory for *)
                              (* REAL or STRING switches *)
                             }

        (* Some convenient lambda-building utilities:  *)
    val pair: LambdaExp × LambdaExp → LambdaExp
    val first: LambdaExp → LambdaExp
    val second: LambdaExp → LambdaExp
    val Let: ((lvar × LambdaExp) × LambdaExp) → LambdaExp
  end
```

Figure 6.6: The signature LAMBDA_EXP

The result of compiling a declaration is a CEnv for the declaration, plus a function which, when applied to a compiled *scope* for the declaration, returns a lambda expression corresponding to the declaration compiled locally for that scope.

There are two reasons for this particular convention. The first is abstraction—an alternative approach would have been to have the compiler translate declarations directly into lambda expressions with some convention for the scope of the declaration (such as a vector of lvars corresponding to the identifiers ordered lexically), but it is more elegant to choose the convention elsewhere, separately from the compiler itself. The second reason for the higher-order nature of `compileDec` is that it makes sequential and local declarations easy to deal with: the functions resulting from compilation of the sub-phrases can just be composed together (as shown in subsection 6.6).

This convention carries down into the functions private to the compiler. Functions which compile expression-like phrases (expressions, atomic expressions and so on) return a `LambdaExp`. Functions which compile declaration-like phrases (declarations, value bindings, and so on) return a function of type `LambdaExp` $\rightarrow$ `LambdaExp`.

The compiler has a simple set of runtime data structures, onto which it must map the runtime data structures (the dynamic semantic objects) of the Definition. The representations are as follows:

- Records become vectors whose elements represent the record fields in lexical order.
- Constructed datatypes become pairs (vectors of two elements), the first of which (element 0) is the argument of the constructor (or `void` for nullary constructors) and the second of which is an integer tag (≥ 0) denoting the lexical index of the constructor in the datatype.
- Exception constructors in the DEnv are held as string references; the string contains the exception's name (the SML identifier) for printing, and the reference models the exception's generative behaviour.
- Exception values and packets have the same representation—a pair whose first element is the value carried by the exception and second element is a *string ref* which embodies the dynamic name of the exception (as the address of the reference) as well as a string for top-level printing.
- Value-carrying value and exception constructors compile into functions which build a data structure of the appropriate form from their argument.

There are two problems with these representations. Firstly, the representation for exceptions means that the expression

```
let
   exception A
   exception B = A
in
   raise B
end
```

results in the uncaught exception `A` being reported at top level, rather than `B`. Secondly, the obvious definition of lists (using a `datatype` declaration) results in two vectors being allocated for each list element. Both problems can be circumvented with a little effort, but this is beyond the scope of this report. The issues involved in the efficient representation of datatypes is discussed further in [1].

We now consider the declaration compiler in more detail. The input to the compiler is a data structure bearing a close resemblance to the abstract syntax of the SML core presented in the Definition (pages 8–9), and is therefore not described here. Note that the abstract syntax of the Definition has been *resolved* such that identifiers are distinguished as variables, data constructors or exception constructors. In addition to this resolution, the compiler needs some type information in order to compile matches and choose data representations in some circumstances; this is discussed in Section 6.7.

6.6.2 Compiling Atomic Expressions

Atomic expressions are compiled as follows. Special constants (integers, reals and strings) have direct counterparts in the lambda language, and are trivially compiled into these without reference to the CEnv. Identifiers are more complicated, since by this stage they have been resolved by the elaborator into one of three classes: variable, data constructor or exception constructor. Variables are looked up in the CEnv yielding lvars (although the pervasives are treated slightly differently, as described in Section 6.8). Nullary data constructors compile into pairs whose first element is `VOID` and second is an integer tag denoting the constructor. When a unary (value-carrying) constructor is applied to a value, a pair is built containing the value and an integer tag for the constructor. The value itself might be a pair (such as arguments to the cons operator, '`::`'); there are more efficient representations for such datatypes, but none has been implemented yet. Each unary constructor is compiled into a function expression which builds a pair when applied to its argument (Figure 6.7). (The code generated for something like `2 :: nil` is, therefore, initially rather inefficient, but the optimiser described in Section 6.9 transforms it into a more sensible form, as shown in the first example of Figure 6.7.)

The compilation of constructors requires information from the elaborator for two reasons. Firstly, the value of the integer tag depends on where the constructor identifier falls alphabetically compared to the other constructors of the datatype. Secondly, value-carrying constructors are detected by their type; if the constructor has a type which is an instance of $\alpha \to \beta$ then it is value-carrying. The compilation of constructors does not require the CEnv, just as the dynamic semantics doesn't require an environment for evaluation of constructors (rule 105).

Source: `let datatype` α `t = T of` α `in T 1 end`

Code: `VECTOR[VECTOR[1, 0]]`

Result: `val it = T 1 : int t`

Source: `let datatype` α `t = T of` α `in T end`

Code: `VECTOR[FN v192. VECTOR[v192, 0]]`

Result: `val it = fn :` $\alpha \rightarrow \alpha$ `t`

Figure 6.7: Code for a unary data constructor

Exception constructors are treated similarly to data constructors, but with the important difference that exceptions are generated dynamically during execution, which complicates matters. Each time an exception declaration is encountered a new exception is generated, distinct from any exception generated from a previous evaluation of the declaration. Each exception is a first-class value, and can be regarded as a function (if the exception is value-carrying, with type $\alpha \rightarrow$ `exn` for some type α) or as an exception packet (if the exception is nullary, with type `exn`). Each exception must, however, be available for pattern matching (regardless of whether it is nullary or unary), and exceptions only match if they are identical at runtime.

The implementation scheme is as follows. A declaration of a new exception will bind an lvar to a new *string reference*, regardless of whether the exception carries a value. The string holds the exception's name for printing. This representation is most convenient for pattern matching, since exception matching then becomes a matter of determining identity of references. When an exception identifier is encountered in an expression, it must be converted into a form consistent with a value of type `exn` or $\alpha \rightarrow$ `exn`. For nullary exceptions, the compiled lambda code pairs the string reference with `VOID`. For unary exceptions, the code builds a closure which takes a value x and returns the pair (x, n) where n is the string reference. Note once again that the aim is to have a unified scheme for application, regardless of the kind of object being applied; this is in contrast to the dynamic semantics of the Definition where there are a number of rules for application.

Records (and tuples) compile to vectors, using a canonical representation where the fields are stored according to the lexical ordering of the field labels. No type information is needed since record expressions are exhaustive in the labels. However, some subtlety is required, since record fields must be stored in canonical order but evaluated in the order in which they occur in the program, in order to maintain determinacy where side effects might be present. This is done by means of a set of local declarations which evaluate the fields in the required order and then build the record in canonical order. Figure 6.8 gives an example where record fields have to be reordered.

```
    Source: {C=f(), B=f(), A=f()};

      Code: LET(v197 = APP(lvar_f, VOID):
                v198 = APP(lvar_f, VOID):
                v199 = APP(lvar_f, VOID):
                VECTOR[v199, v198, v197]
               )
```

Figure 6.8: Code generation for records

Atomic expressions of the form `let` *dec* `in` *exp* `end` cause the declaration to be compiled, yielding a CEnv ce_1 for the declaration and a function f to be applied to the scope, which is the result of compiling *exp* in the original CEnv augmented with ce_1.

6.6.3 Compiling Expressions

Expressions are compiled in the context of a CEnv, yielding a lambda expression, as follows. Function application is fairly simple, because the type of the functional part is not discriminated at application time; instead, the compiler ensures that it will always be a closure. However, it is here that we deal with pervasive functions, using a mechanism described in Section 6.8.

Function (`fn`) expressions are dealt with directly by the match compiler (Section 6.7). Exception handling is done at runtime by a meta-circular interpreter (one which implements exceptions using exceptions): `raise` expressions are compiled into `RAISE` lambda terms and `handle` expressions are compiled into `HANDLE` lambda terms, which are implemented in the lambda interpreter with exceptions. In the case of `HANDLE`, the lambda term contains a function `FN` sub-term built by the match compiler.

6.6.4 Compiling Declarations

The function `compileDec` forms the top-level of the compiler; since we don't currently support the modules system, phrases passed to the compiler correspond to core declarations. `compileDec` has type

    CEnv → dec → (CEnv × (LambdaExp → LambdaExp))

as explained in Section 6.6.1. The subsidiary functions for dealing with valbinds, exbinds and so on have similar types.

We shall dismiss the simple cases first. Type declarations play almost no part in SML's dynamic semantics, and so the compiler can ignore their effect[3]. Therefore, `compileDec` compiles `type` and `datatype` declarations into an empty CEnv together with the identity

[3]We say *almost* no part because there is an obscure situation, highlighted in the SML Commentary [10], which requires `datatype` declarations to have an effect at evaluation time. However, this situation occurs in connection with the modules system which the Kit currently does not implement.

function. Empty declarations and infix declarations[4] are treated similarly. For phrases of the form

`abstype` *datbind* `with` *dec* `end`

the *datbind* is discarded. `local` and sequential declarations are similar to each other; they differ only in the CEnv produced as a result. In both cases, the first declaration is compiled yielding a CEnv ce_1 and result function f_1. The second declaration is compiled in an environment augmented with ce_1, yielding ce_2 and f_2. The resulting CEnv is $ce_1 + ce_2$ (for sequential declarations) or ce_2 (for `local` declarations); in both cases, the result function (of type `LambdaExp` $\rightarrow$ `LambdaExp`) is $f_1 o f_2$.

Some complexity arises when dealing with value bindings and exception bindings. We deal with exception bindings first.

Exceptions require rather sophisticated compiler support, for a variety of reasons. Exceptions are generative: if an exception declaration serves to introduce a new exception (as opposed to a new identifier binding for an existing exception), then each execution of that declaration creates a new, distinct, exception. (By contrast, datatype declarations compile to no code.) Exceptions may be nullary or they may carry values, in the same way as constructors. And, exceptions may be pattern-matched, in which case they must be matched sequentially according to their dynamic identity, rather than identifier or textual occurrence. By contrast, data constructors are not generative, and may not be renamed without losing their constructor status; hence, constructors in patterns may be distinguished merely by their identifiers.

The representation of exception values and packets is described in Section 6.6.1. However, the dynamic environment (DEnv) holds exceptions simply as string references. This makes pattern matching over exceptions straightforward; if they were held in the same representation as values of type `exn`, then extra dereferencing would result, and value-carrying exceptions would have to decomposed from their closure representations.

When an exception is encountered in the context of an expression, therefore, it is converted into a value of type `exn` (which is a vector of length two) or a value of type $\tau \rightarrow$ `exn` for some τ (which is a closure).

We can now consider exception declarations in detail. A declaration of the form

```
exception A
```

causes the identifier `A` to be bound to a new lvar in the CEnv when the declaration is compiled; the lvar is bound to `ref("A")` in the DEnv each time the declaration is executed. A value-carrying exception is treated identically. A declaration of the form

```
exception B = A
```

introduces a binding for `B` to `A`'s lvar in the CEnv at compile-time, and leaves the DEnv unchanged at runtime (and in fact, no code is generated). Compound declarations using `and` are straightforward.

[4]Infix declarations find their way into the abstract syntax, although they are omitted from the semantic rules of the Definition.

An actual exception value is built when an exception identifier is encountered in the context of an expression. It is here that value-carrying exceptions are distinguished from nullary exceptions. Consider the declaration

```
exception A1
      and A2 of int
```

The declaration produces the bindings

`A1` $\rightarrow lv_1$
`A2` $\rightarrow lv_2$

in the CEnv and

$lv_1 \rightarrow$ `ref "A1"`
$lv_2 \rightarrow$ `ref "A2"`

in the DEnv. If identifier `A1` is encountered in an expression, the code generated retrieves the string reference and builds a vector. If the identifier `A2` occurs, the code returns a closure for a function which pairs its argument with the string reference.

We now describe the other kinds of declaration. Recursive function declarations are straightforward; SML disallows any other kind of object to be declared recursively, and even places syntactic restrictions on the functions (recursive declarations may not, for instance, bind variables to expressions which just happen to have functional type).

Within a `rec` declaration (of which `fun` is just a syntactic derived form), there may only be function bindings and further, nested, occurrences of `rec` (which serve no purpose). The entire topmost `rec` declaration is flattened out to yield a set of mutually recursive function bindings, which are then translated into a `FIX` declaration in the lambda language.

The recursive declaration is compiled in two passes. The first pass assembles the function identifiers to be bound and creates new lvars for them. The second pass compiles the declaration's right hand sides (which must all be lambda expressions) in the outer CEnv plus the CEnv for the function identifiers. The resulting lvars and compiled functions are then assembled into a `FIX`.

Finally we consider plain value bindings. These are actually the most complicated in terms of the code generated, because value bindings may contain patterns and must therefore use the match compiler.

An expression of the form

`let` *pat* `=` *exp* `in` *scope* `end`

is regarded as the equivalent

`(fn` *pat* `=>` *scope*`)` *exp*

where the lambda expression contains a *match*. Value bindings are therefore dealt with by the machinery responsible for function expressions and applications.

6.7 The Match Compiler

The whole area of efficient pattern matching is quite a complex one. The match compiler implemented for the Kit is fairly sophisticated, and beyond the scope of this paper, but it is at least necessary to describe what it does, and how it interfaces with the rest of the compiler. We begin with an overview of the semantics of pattern matching in SML.

The Definition describes a syntactic construct called a *match*, which is a list of (pattern, expression) pairs. Matches occur in many of the contexts where identifiers are bound (specifically, case expressions, function arguments and exception handlers), and it is convenient to interpret non-recursive value bindings in a similar manner.

In fact, case expressions can be ignored since they are nothing more than convenient sugaring for function application. Similarly, we can ignore "formal" function declarations using `fun`, including curried functions, since they are just shorthand for (recursive) value bindings to lambda expressions.

Consider the expression

```
fn (0, _) => e1
 | (_, 0) => e2
 | (_, _) => e3
```

This is an anonymous function which takes a pair of integers as argument. The SML semantics requires that it attempt to match its actual parameter against the patterns in sequence, so that calling it with argument `(0, 0)` would return the result e_1, even though all three patterns match the actual argument. This particular function employs a match which is *exhaustive*, in that every possible actual parameter is matched by some pattern (in fact, `(_, _)` will match any argument which typechecks). The match is also *irredundant*, since each pattern in the match will be chosen for some argument. Inexhaustive and redundant matches are legal, but the Definition requires that each must result in a warning message from the compiler.

We choose to treat all non-recursive `val` bindings as instances of pattern matching. Each is an instance of the phrase class

```
val pat = exp
```

A simple declaration such as

```
val x = 3
```

performs pattern matching on the pattern consisting of the variable `x`. The compiler, parameterised on a scope for the declaration, transforms it into the form

```
(fn x => scope) (3)
```

at which point it can use the match compiler for the `fn` expression.

The Definition gives a formal, operational, definition of pattern matching. A value can be matched against a pattern in the context of an environment (which is needed only to give meaning to exception constructors in the pattern: see the Definition, rules 146 and 147). The match either succeeds, yielding a value environment for the variables in the pattern, or fails. When evaluating a match, failure causes execution to pass to the next rule so that the next pattern can be matched against the value. When evaluating a value binding, failure to match causes an exception (`Bind`) to be raised.

Although perfectly acceptable for a semantic definition, this operational, backtracking, view is inefficient from a practical point of view, for a number of reasons which are best illustrated by examples (in which we assume the declaration of a datatype providing constructors `RED`, `ORANGE`, `YELLOW` and so on in the obvious way).

Consider the following:

```
fn (1, 2, (3, [RED, GREEN, RED])) => e1
 | (1, 2, (3, [RED, GREEN, GREEN])) => e2
 | (1, 2, (3, [RED, GREEN, BLUE])) => e3
```

Executing this match in the sequential, backtracking manner suggested by the semantics would be inefficient. The patterns are largely identical, differing only in the third element of a list expression which is nested quite deeply in the patterns, and sequential matching might well cause the argument value to be decomposed several times, with largely identical tests being performed each time.

Even in examples with minimal backtracking, the sequential nature of the Definition's matching process precludes any efficient case analysis over data constructors. In the example

```
fn RED => e1
 | ORANGE => e2
 | YELLOW => e3
 | GREEN => e4
 | BLUE => e5
 | INDIGO => e6
 | VIOLET => e7
```

there is little backtracking involved, yet sequential matching is still inefficient compared to a single linear switch.

The Definition decomposes record patterns from left to right, but this is essentially a notational convenience. Since SML is a strict language, there is no obligation to do left-to-right pattern decomposition, and it may be advantageous to choose a different order. In terms of the number of comparisons performed for an arbitrary data value, there is no overall optimal order in which to decompose the tuples of this match—different orders will be optimal for different data values. A more practical aim might be to minimise the number of decision nodes in the code generated for a match; in this case, the problem can be shown to be NP-complete [6]. In practice, we eliminate backtracking (example 1) and identify linear switches (example 2), and adopt a simple scheme to choose the most appropriate first match in a record. The details are beyond the scope of this paper.

The scheme adopted for compilation of matches and bindings is based on the concepts behind an earlier effort by Marianne Baudinet [6], although that code was written in Lisp and not referred to for the current implementation. The scheme involves the generation of *decision trees*. A decision tree is an intermediate data structure which encompasses the decompositions and selections necessary to determine, for any data value, the topmost pattern which matches that value. (We must, of course, provide the same effect as the top-to-bottom, backtracking matching of the semantics.) For any data value, a decision tree performs an analysis which either delivers the first matching rule, or indicates a failure (if the value matches no rule). If a rule is matched, the decision tree also delivers the *value environment* for the pattern. This is a CEnv mapping the pattern's variables to lvars corresponding to internal nodes of the argument.

The generation of decision trees is beyond the scope of this paper; however, we shall describe the trees themselves and show some examples. The `DecisionTree` data structure is shown in Figure 6.9. It should be noted that decision trees are *typed*, in that they contain entities such as constructors and record labels whose interpretation (as far as the code generator is concerned) is dependent on their type. Decision trees carry lvars which are relevant to the lambda code generator. In addition, decision trees are *symbolic* in that they refer to data decompositions in an abstract sense (label decomposition, constructor decomposition, and so on) rather than in terms of the concrete data operations (specifically, `SELECT`) of the lambda language.

The rôle of a decision tree is to decompose and examine an argument value to determine whether it matches a specific pattern, and if so to return the CEnv binding the variables of the pattern to the appropriate sub-components of the argument. A decision tree contains nodes which perform decompositions of records and constructors, as follows:

`LAB_DECOMPOSE`{`bind, parent, lab, child`} extracts a field `lab` from a record `parent`, binding it to a new lvar `bind`;

`CON_DECOMPOSE`{`bind, parent, child`} decomposes a constructed value `parent`, binding the value to which the constructor was applied to the lvar `bind`;

`EXCON_DECOMPOSE`{`bind, parent, child`} performs a similar operation for exception constructors;

`CON_SWITCH`{`arg, selections, wildcard`} performs a case selection over a set of constructors, depending on the constructor of the constructed value bound to `arg`. If the set is exhaustive then the wildcard is omitted, otherwise the wildcard is the decision tree to be traversed if the value matches none of the constructors present;

`SCON_SWITCH`{`arg, selections, wildcard`} performs a case selection over *special constants* (integers, reals or strings). The wildcard must always be present since a selection over constants can never be verified as exhaustive at compile-time. (Recall from Section 6.5 that the wildcard is optional in the lambda language's `SWITCH_I` instruction, since this can be generated by the compiler from `CON_SWITCH` decision nodes. An `SCON_SWITCH` corresponds to constant patterns in the source code, and these can never be exhaustive.)

`EXCON_SWITCH`{`arg, selections, wildcard`} performs a case selection over exception constructors. Note that the selection is *sequential* (a list rather than a map). This

```
signature DECTREE_DT =
  sig
    type lab and lvar and var and con
    type longexcon and scon and pat
    type type_info
    type RuleNum sharing type RuleNum = int
    type (α₌, β) map
    type CEnv

    datatype α option = NONE | SOME of α

    datatype DecisionTree =
      LAB_DECOMPOSE of {bind: lvar, parent: lvar, lab: lab,
                        child: DecisionTree, info: type_info
                       }

    | CON_DECOMPOSE of {bind: lvar, parent: lvar, child: DecisionTree}
    | EXCON_DECOMPOSE of {bind: lvar, parent: lvar, child: DecisionTree}

    | CON_SWITCH of {arg: lvar,
                     selections: (con, (type_info × DecisionTree)) map,
                     wildcard: DecisionTree option
                               (* An option because we may notice that all *)
                               (* the constructors are present.  *)
                    }
    | SCON_SWITCH of {arg: lvar,
                      selections: (scon, DecisionTree) map,
                      wildcard: DecisionTree
                     }
    | EXCON_SWITCH of {arg: lvar,
                       selections: (longexcon × DecisionTree) list,
                       wildcard: DecisionTree
                      }

    | END of {ruleNum: RuleNum, environment: CEnv}
    | FAIL
  end
```

Figure 6.9: The decision tree data structure

is because it is impossible to determine the identity of exceptions at compile-time (cf. value constructors, whose values are, semantically, simply the identifiers). Consider the following example:

```
let
   exception E1 of int
   exception E2 = E1
in ... handle E1(0) => e₁
            | E2(n) => e₂
            | E1(n) => e₃
end
```

It is incorrect to merge the two occurrences of `E1` in the match since `E2` is dynamically identical to `E1` and so a packet `E1(1)` matches `E2(n)`.

(It might be argued that the compiler could maintain an environment to track the identity of exceptions in cases such as this. However, examples using the modules language illustrate that exception identity cannot always be determined at compile-time, and we wish to encompass the modules language at some stage.)

`END`{`ruleNum, environment`} indicates a successful match and provides the index `ruleNum` of the matching rule together with a CEnv `environment` for the variables of the appropriate pattern. An `END` node does not contain any vestige of the right-hand-side of the match rule, since the right-hand-sides may not exist, as explained below.

`FAIL` indicates the failure of all patterns to match the value.

The decision tree contains some embedded type information provided by the type-checker. The compiler must convert the labels and constructors of the decision tree into integer selections in the lambda language, and therefore needs to know the ordinal index of each constructor and record label. In addition, the type information is needed to determine the exhaustiveness of a value constructor switch; it allows the compiler to determine whether or not all the constructors are present.

The compile-time warnings (irredundancy, exhaustiveness) are performed in a separate post-pass over the decision tree. The unreachable rules of a match are those which don't occur in any `END` node. A match is inexhaustive if the decision tree contains any `FAIL` nodes.[5]

The final stage in compiling a match is to generate lambda code from the decision tree. This is done by a function with the following specification:

```
fun compileDecisionTree (env: CEnv)
                        (tree: DecisionTree,
                         compiler: (int × CEnv) → LambdaExp,
                         failure: LambdaExp
                        ): LambdaExp
```

[5]The assumption here is that unreachable `FAIL` nodes are never generated in our decision tree building algorithm. Experimentation suggests that this is the case, but it is not yet proven.

`compileDecisionTree` takes a CEnv as (curried) argument. This is only needed to generate code for case selection over exception constructors: the identity of an exception constructor depends on the dynamic environment. The `failure` argument is the lambda expression to plant for each `FAIL` node: an expression is chosen to raise the pervasive exception `Match` in lambda expressions, `Bind` in value bindings, and will re-raise the root of the match in exception handlers. Of most interest is the higher-order argument `compiler`. This is an encapsulation of the compiling function to be applied to the right hand sides of the match. Recall that decision trees are generated from a root lvar and a list of patterns only—the right hand side expressions are not provided. This is for the very good reason that they might not exist: consider the case of a top-level binding such as

```
    val (x, y) = (1, 2)
```

This is passed to the match compiler as

(fn (x, y) => *scope*) (1, 2)

where *scope* is lambda-abstracted in the compiler. So, `compileDecTree` is called with an encapsulation of the expression compiler which delivers the lambda code for the appropriate right hand side. The compiler is passed the rule number and the CEnv for the variables in the pattern.

It should be noted that a rule might occur in several leaves of a decision tree. In such cases, it is possible to lambda-abstract the right hand side in order to avoid the duplication of code. The abstraction would be performed over the variables of the pattern. This optimisation is not performed at present.

Compilation from decision tree to lambda code is fairly straightforward. All the decomposition operations compile to the appropriate kind of `SELECT`. `CON_SWITCH` compiles to an integer switch over the tag field of the argument. `SCON_SWITCH` compiles directly to the appropriate kind of `SWITCH` in the lambda language. `EXCON_SWITCH` is slightly more complicated, since the matches must be performed in sequential order. The code generated consists of a nested sequence of conditional expressions: the argument value's exception reference is compared against each exception constructor in turn, using the pervasive equality predicate over the string references.

6.8 The Primitive Functions

Any Standard ML system must provide the built-in functions described in appendices C and D of the Definition. Some of these (such as `map` and `rev`) can be defined directly in SML. Others (such as the arithmetic operators and the equality predicate) cannot. The Definition defines the latter in two ways: the initial static basis contains the types of the pervasives, and the initial dynamic basis contains the basic values which are differentiated at application time (rule 116). The special status of the basic values is inelegant in a compiler, partly because the contents of several environments have to be "hard-wired" into the compiler in several places, and also because we have decided that the different kinds of function application should be differentiated at compile-time rather than runtime.

Therefore, we choose to have a single built-in function called `prim`, and we define the other pervasives in terms of it.

The initial static environment contains a single entry: the identifier `prim`, with type `(int` $\times$ α`)` $\rightarrow$ β. In order to describe the dynamic binding of `prim`, we must revise our description of compiler environments (CEnv's). The interface `COMPILER_ENV` contains the specifications

```
datatype result = LVAR of lvar
                | PRIM

val lookupLongvar: CEnv → longvar → result
```

and the initial compiler environment just contains a mapping from `prim` to `PRIM`. Whenever identifiers are declared in a CEnv, they map to `LVAR(`*lv*`)` for some *lv*.

The compiler looks out for the special case where an identifier mapping to `PRIM` is applied directly to an argument. (Looking for textual occurrences of `prim` is inelegant, and would require redeclaration of `prim` in any way to be prohibited.) In such an application, the argument is required to be a record with two fields, the first of which must be an integer constant. This number must coincide with the implementation of the semantic function APPLY which in the compiler takes an integer index. Occurrences of `PRIM` are converted into `PRIM_APP` in the lambda language. Because the application of `PRIM` is detected and converted in the compiler, the DEnv has no knowledge of it, and no lvars are involved.

`PRIM_APP` always takes two arguments. The first is an integer index indicating the primitive function required, and the second is a either a single argument or a tuple of arguments for the primitive function. Thus, the type of `prim` can be restrained to `(int` $\times$ α`)` $\rightarrow$ β rather than the more general (and less safe) $\alpha \rightarrow \beta$. If a primitive function is capable of raising an exception, the exception packet or function is passed as one of the arguments; hence, there are no reserved exceptions in the implementation of APPLY.

The pervasive functions are defined in a *prelude*, part of which is shown in Figure 6.10. This prelude is processed by the Kit when it is built. Each pervasive identifier is bound to an application of `prim` to the correct arguments, type constraints being added to constrain the unsafe polymorphic type of `prim` to the correct type for the primitive in question. By convention, equality is primitive 0, and is defined in the prelude in the same way. (This perhaps violates the assertion in the Definition that the equality symbol may not be rebound; however, we would argue that this is its *initial* binding.) Parts of the compiler which require the equality predicate generate lambda code containing `PRIM_APP(0, ...)` directly. This implementation scheme means that the primitives are implemented in a rather inefficient manner (there is an extra level of function application for each primitive application), but we would rather defer that to later optimising passes, and keep the basic scheme simple.

```
val (op =) = fn (x: α₌, y: α₌) => prim(0, (x, y)): bool
val floor = fn (x: real, y: real) => prim(1, (x, y, Floor)): int
val real = fn (x: int) => prim(2, x): real
val sqrt = fn (x: real) => prim(3, (x, Sqrt)): real
val sin = fn (x: real) => prim(4, x): real
val cos = fn (x: real) => prim(5, x): real
...
```

Figure 6.10: Part of the initial prelude

6.9 Lambda Code Optimisation

It might seem strange to implement an optimiser in a compiler which makes no pretence at being efficient. The reasons for implementing an optimiser for the lambda language are two-fold. Firstly, the earlier stages of the compiler (the decision tree generator in particular) generate a lot of garbage, such as bindings of variables which are never used. In addition, simple constructor applications such as `CON(3)` are compiled into function applications with a closure for `CON`, so we should a least implement an optimiser to remove these. It is therefore desirable to do simple optimisations in order to clarify the final code for the purposes of inspection. Secondly, the optimiser was an afternoon's work, and an interesting exercise.

The interface to the optimiser is shown in Figure 6.11, and its function is fairly self-evident. Internally, it is implemented in two parts. There is a generic tree-walker which walks over a lambda expression, applying a `LambdaExp` → `LambdaExp` function to each node, and in addition there is a set of simple optimising functions, each of which has type `LambdaExp` → `LambdaExp` and operates just on the node which is passed to it. The optimiser operates by building the composition of all the optimising functions, and calling the tree-walker with this composition. The optimiser strives for a fixpoint at which none of the individual optimisations can be performed anywhere over the lambda expression. Each optimising function touches a reference variable whenever it succeeds in applying its optimisation; the optimiser makes repeated passes over the input until a pass is made which leaves the reference untouched.

We describe the optimisations below, but must first define some terms. A lambda sub-expression is said to be *small* if it can be substituted for a variable occurrence without a large increase in code size. A sub-expression is said to be *safe* if it performs no side-effects. The leaf nodes denoting variables and constants are considered small. The conditions for safeness are, in theory, more complex, but the algorithm used is a simple-minded one: lambda terms are safe if they contain no applications or pervasive applications, and don't raise exceptions. Safe terms are those which can be removed or relocated (within the same or a nested context) without changing the meaning of the program.

It is here that the uniqueness of lvars is especially useful. Lambda terms can be

```
signature OPT_LAMBDA =
  sig
    type LambdaExp

    val optimise: LambdaExp → LambdaExp
  end
```

Figure 6.11: The signature `OPT_LAMBDA`

examined for specific lvars without consideration of scope, since there is only one binding of each lvar. And, terms can be moved from outer to inner contexts without variable renaming being necessary, since no inner context can rebind a variable referred to by an enclosing term.

The optimisations are as follows:

UNUSED: An lvar binding can be removed if the lvar is not used in the scope of the binding, *and* the lvar is bound to a safe term.

BETA1: If an lvar is bound to a small, safe term, each occurrence of the lvar in the scope is replaced by the term. When the small term is another variable, this implements variable renaming.

BETA2: If an lvar is bound to a safe term (of any size) and the lvar is only used once in the scope, the binding is removed and the lvar replaced with its term.

HOISTFIX: By default, "`fun`" declarations compile into `FIX` lambda code constructions. This optimisation lifts non-recursive function definitions out of the `FIX`, making them available for beta optimisation.

FIX0: The HOISTFIX optimisation might leave behind empty `FIX` constructors; FIX0 removes them.

SWITCH: The match compiler can, in some circumstances, produce switches with a wildcard but no discriminants. This optimisation replaces each such switch directly with its wildcard.

Beta reduction is not performed over functions. If it were, variable renaming would be necessary to avoid duplication of binding occurrences of lvars, and steps would have to be taken to deal with the Y combinator.

6.10 Execution

Recall that `compileDec` has type

```
CEnv → dec → (CEnv × (LambdaExp → LambdaExp))
```

which allows the compiler to deal with declarations rather than expressions. At top-level, each core declaration is passed to `compileDec`, to return a CEnv for the declaration plus a function f of type `LambdaExp` $\rightarrow$ `LambdaExp`. From the CEnv is built a `VECTOR` lambda expression containing all the lvars in some canonical order, and f is applied to this. The result is a lambda expression for the top-level declaration over a scope containing a tuple of the variables bound by the declaration. This lambda expression is then executed, to return (if successful) a vector of object values, one for each variable. The vector is then decomposed to form a DEnv for the declaration. Finally, the top-level basis is augmented with the CEnv and the DEnv.

The top-level context of the compiler is a `DynamicBasis`, comprising a CEnv (which maps all the visible top-level identifiers to lvars) and a DEnv (which maps all the lvars for the visible top-level identifiers to their runtime representations[6]). Each top-level declaration is compiled in the top-level CEnv and executed in the top-level DEnv, and the new bindings are added to the top-level if it terminates successfully.

The interface to the lambda interpreter is shown in Figure 6.12, and is fairly self-explanatory. In practice, the implementation is fairly simple as well. Lambda expressions are passed to a function

```
run: DEnv → LambdaExp → object
```

which executes them in a DEnv representing the current top-level environment. Variables in the lambda expression are retrieved directly from the DEnv. Constants evaluate directly to the corresponding objects. Function (`FN`) expressions evaluate to closures encapsulating the current environment. The treatment of `FIX` is similar to the treatment of `val rec` in the Definition: firstly, the function expressions which form the right hand sides of the fix are evaluated in the current environment, yielding closures which do not incorporate the identifiers in the recursive bindings. Then, an environment is built which maps the identifiers to the closures. Finally, the Rec operation is applied to the environment—this traverses the closures, replacing the recursive environment component of each closure with the entire environment, including the `FIX`-bound variables. This provides for a single level of recursion; indefinite recursion is provided for by subsequent applications of Rec within function applications. The operation of Rec is described more fully in the SML Definition and the SML Commentary.

Application (`APP`) of one lambda expression to another results in the function and argument being evaluated, and then the body of the function being executed in the closure environment of the function extended by an lvar binding for the argument. Calls to the pervasive functions (`PRIM_APP`) result in calls to the `Apply` module on the evaluated argument. `VECTOR` and `SELECT` produce calls to `Objects.vector` and `Objects.select`.

Exception handling is done in a meta-circular manner; the lambda interpreter contains a single exception called `PACKET`. When a lambda expression raises an exception, the interpreter raises `PACKET` with the exception packet. Expressions of the form `HANDLE(`e_1`,` e_2`)` cause e_1 to be interpreted in a context which handles `PACKET`, in which case e_2 (which

[6]It is assumed that when SML variables go out of scope at top level, their lvars are marked for garbage collection.

```
signature RUN_LAMBDA =
  sig
    type DEnv
    type LambdaExp
    type object

    exception UNCAUGHT of string

    val run: DEnv → LambdaExp → object
  end
```

Figure 6.12: The signature RUN_LAMBDA

must be a FN lambda term) is applied to the packet's value. If PACKET propagates to the top level it is transformed into an exception called UNCAUGHT so that it can be reported at top level (or signalled into an enclosing use context).

The implementation of SML references and assignment (i.e. the constructor ref and the function :=) is currently incomplete, and so we defer a discussion of references to a future paper.

6.11 Conclusions

We have attempted to show that it is possible to build an SML to intermediate lambda language compiler which is modular and which uses simple data structures. Our thesis is that modularity and abstraction are initially more important than raw efficiency—a well-structured compiler can be made more efficient by optimising the modules and refining the interfaces between them, but an efficient compiler can rarely be improved in modularity and structure.

The lambda language is small and simple, with nice properties (such as the unique identity of lvars) and yet it is a full functional language in its own right. Currently, many projects depend on making alterations and enhancements to the SML source text or treating it in some special way; we hope that the lambda language can be used in this way instead, thus avoiding the complexity and ambiguities of the SML language itself.

The match compiler is largely complete but only contains trivial optimisation heuristics. We would like to develop it further and make comparisons of heuristics. Since we are claiming modularity as a virtue of the Kit, it should be easy to replace the current match compiler with others; an implementation of Wadler's scheme [9] would be a useful exercise, as well as a "trivial" sequential implementation (in the style of the semantic rules), in order to compare code size and performance directly.

Chapter 7

Miscellaneous

7.1 Introduction

This document covers a number of assorted design topics involving the ML Kit. These issues are sufficiently small that they do not merit reports of their own, and yet it is important that they be covered in order to gain a more complete understanding of how the Kit operates. The sections are as follows:

Section 7.2 covers the evaluation of the Modules language and the implementation of the associated dynamic environments;

Section 7.3 describes some of the higher levels of the interpreter, and the way that the execution and evaluation heirarchy is structured;

Section 7.4 describes the mechanism used for top-level printing;

Section 7.5 details the implementation of the pervasive functions of the Kit, corresponding to APPLY in the Definition;

Section 7.6 describes the detection and reporting of elaboration errors;

Section 7.7 describes file nesting and inclusion, and the semantics and implementation of the `use` function;

7.2 Evaluation of the Modules

`EvalTopdec` is the evaluator for the Modules language. Its signature (`EVALTOPDEC`) is shown in Figure 7.1. This signature is common to both the Interpreter and the Compiler, since they share the same interface; `EvalTopdec` is specific to the Interpreter. (The Compiler has its own implementation.) Top-level declarations are evaluated in a basis, yielding a new basis just as in rules 191 to 193 of the Definition. Any exception raised in the evaluation which propagates to the top level causes the meta-exception `UNCAUGHT` to be raised with a packet. (The exception `UNCAUGHT`, as well as the type `Pack` and the other exception manipulation facilities, are actually propagated from `EvalDec`.) `EVALTOPDEC` also contains pretty-printing functions for printing dynamic bases.

```
signature EVALTOPDEC =
  sig
    type topdec
    type DynamicBasis
    type Pack

    val RE_RAISE: Pack -> unit

    exception UNCAUGHT of Pack
    val pr_Pack: Pack -> string

    val eval: DynamicBasis * topdec -> DynamicBasis
    val FAIL_USE: unit -> unit

    type StringTree
    val layoutDynamicBasis: DynamicBasis -> StringTree
  end
```

Figure 7.1: The signature EVALTOPDEC

The other functions in EVALTOPDEC (RE_RAISE and FAIL_USE) relate to the implementation of the built-in function use, and are documented in Section 7.7.

The EvalTopdec functor takes as arguments the abstract syntax (signature TOPDEC_GRAMMAR), the dynamic objects of the Modules (signature MODULE_DYNOBJECT), the Core level evaluation machinery (EVALDEC and CORE_DYNOBJECT), and other utility structures. The top-level grammar is resolved such that identifier classes are known; this must be the case because EvalTopdec requires that the declaration type (dec) of TOPDEC_GRAMMAR shares with that of EvalDec, which requires a resolved grammar.

The implementation of Modules execution (functor EvalTopdec) follows fairly directly from the rules in the Definition. The topmost function eval corresponds directly to the final three Modules evaluation rules of the Definition (191 to 193). The lower level functions are mutually recursive, and correspond to the rest of the semantic rules, as follows:

evalStrexp (structure expressions) for rules 160–163;

evalStrdec (structure declarations) for rules 164–168;

evalStrbind (structure bindings) for rule 169;

evalSigexp (signature expressions) for rules 170 and 171;

evalSigdec (signature declarations) for rules 172–174;

`evalSigbind` (signature bindings) for rule 175;

`evalSpec` (specifications) for rules 176–183;

`evalStrdesc` (structure descriptions) for rule 186;

`evalFunbind` (functor bindings) for rule 187;

`evalFundec` (functor declarations) for rules 188–190.

Rules 184 (value descriptions) and 185 (exception descriptions) are essentially dealt with "in line" in `evalSpec`, using the operations `Vars_in_Int` and `Excons_in_Int` (from `ModuleDynObject`) respectively. Rules 191–193 are rolled into the main function `eval`. The Inter and ↓ operations are implemented in `ModuleDynObject`.

7.3 Linking

This section describes the very top level of the system, and the way that the execution and evaluation levels are linked together. These levels are above the Modules evaluator described in Section 7.2, and are in fact common to both the "pure" interpreted system and the lambda compiler. Chapter 6 gives more insight into the way that the execution levels for the compiler and interpreter are abstracted so as to appear identical to the rest of the system; we will only discuss this abstraction where it relates to the rest of the system.

The top-level of the system is a functor called `KitCompiler`. It takes as argument a string which is the name of a file to read as the *prelude* (Section 7.5); it has no other arguments. It returns a structure with the signature given in Figure 7.2. `KitCompiler` is also responsible for "tying the knot" in order to implement the `use` function, in the manner described in Section 7.7.

The main constituents of `KitCompiler` are `Linking`, which brings together the lower levels of the Kit together with the parsing, elaboration and evaluation levels, and `Interpreter`, which implements the top-level loop and some rudimentary file redirection. In its turn, `Linking` calls on the highest level functors for parsing (`TopdecParsing`), the static semantic objects of the language (`StaticObjects`), the elaborator (`Elaboration`) and the evaluation stage (`Execution`).

At this stage we should explain that `Elaboration` and `Evaluation` are regarded as *linking functors*, and form part of the "spine" which ties together the entire Kit. The spine functors live in the file `KitCompiler.sml`, and build various levels of the Kit (`Tools`, `Basics`, `TopdecParsing`, `StaticObjects`, `Elaboration`, `Execution`, `Linking`, `KitCompiler`). The style of these functors differs from those which comprise the body of the Kit proper: the linking functors refer to the functors of the Kit freely, and each linking functor refers to the linking functor in the level below it. Each linking functor contains substructures obtained by applying the linking functors below it; at each level, access might be required to several stages of the linkage, and the functors cannot be applied each time without the resulting types and structures violating the sharing constraints required by the rest of the Kit.

There are also other linking functors which do not form a part of the main spine. The lambda compiler contains a small number of linking functors (one for the top level of the compiler and one for the match compiler), and `Evaluation` is also a linkage stage. This is because it is at the level of `Evaluation` that the Compiler and Interpreter versions of the Kit diverge. There are two `Evaluation` functors, with the same result signature and the same set of arguments (although neither functor is required to use all of them). Each `Evaluation` functor is a linkage functor for its respective subsystem. The Compiler's `Evaluation` functor is described in the compiler Chapter 6. we consider only the Interpreter version here.

In fact, the internal structure of `Evaluation` is simple. It draws together the elements needed for Modules-level evaluation (basic and special values, dynamic objects for the Core and Modules, evaluators for Core and Modules, and top-level printing). It exports a structure for the dynamic basis (signature `DYNAMIC_BASIS`), a top-level declaration evaluator (`EVALTOPDEC`) and top-level value printing (`VAL_PRINT`)—these are the three elements whose implementation differs between the Compiler and the Interpreter.[1] We describe `ValPrint` in Section 7.4.

The signature `INTERPRETER` is shown in Figure 7.3. The functionality provided is roughly equivalent to the interface provided by `KitCompiler`. `Interpreter` performs its own stream handling (indirectly, via the operations provided by `Parse`). It provides the persistent top-level basis (all the lower levels of the system are "pure" in that they take environmental objects as arguments and return new, incremental ones as results). It initiates parsing, elaboration and execution, and detects and reports errors in any of the three phases.

7.4 Top-level Printing

Any SML system needs a way of reporting top-level bindings to the user. This involves two processes: the printing of dynamic values, and the printing of static environments and types. Some of the entities to be presented to the user are, for our purposes, purely static in nature. For example, signatures and functors are reported merely as static objects; their dynamic (runtime) representation is not used. Other entities contain both a static and a dynamic component: most core-level value bindings, for example, are reported as a combination of value and type. (Those which are not include functions, and values with abstract types.)

The printing of values is not simply a case of presenting the static and dynamic components in a neat and tidy way: for example, the dynamic component might be irrelevant, and the static component might need to be reorganised in order to print it in the way that the users wants to see it, which usually corresponds to the source language syntax (an obvious example is functor bindings).

Value printing is driven at the top-most level by the traversal of a basis, which contains

[1]In fact, `ValPrint` should be common between them, and use a lower-level runtime value traversal module with a common interface between the Compiler and the Interpreter. At present, `ValPrint` just inspects the runtime representation directly, with no reference to the static types of values.

```
functor KitCompiler(val prelude: string)
   : sig
      val parse: unit -> unit
      val elab: unit -> unit
      val eval: unit -> unit

      val parseFile: string -> unit
      val elabFile: string -> unit
      val evalFile: string -> unit
   end =
struct
   ...
end
```

Figure 7.2: The functor `KitCompiler`

```
signature INTERPRETER =
   sig
      datatype Mode = PARSE_ONLY | ELABORATE_ONLY | EVALUATE

      val interpretStdIn: Mode -> unit -> unit
      val interpretFile: Mode -> string -> unit
      val interpretString: Mode -> string -> unit
   end
```

Figure 7.3: The signature `INTERPRETER`

a static basis and a dynamic basis (as described in the Definition), as well as the infix basis. `TopLevelReport.report` takes a basis as argument, along with a flag indicating whether dynamic values should be printed as well: if the Kit is being run purely as an elaborator, then there will be no values to print. The result of printing a top-level basis is a *report* (as described in Section 7.6), which is a representation of the lines to be printed. The report is generated by a traversal of the infix basis (in order to report infix bindings) onto which is appended a traversal of the static and dynamic bases.

In theory (and assuming that both bases are present), the printout is driven by a traversal of the static basis and the dynamic basis in parallel. The traversal is a recursive one over both data structures, although the dynamic basis is ignored for the upper levels of the language (since the Modules objects are not printed as dynamic values at all). The dynamic basis is used when the data values of the core are reached: most of the core values need to be printed as a (dynamic) value with a (static) type.

In practice, this parallel traversal is decoupled and the printout is driven by a traversal of the static basis, with the dynamic basis being traversed at the leaves of the static basis. This is a simpler solution because dynamic basis traversal is sufficiently simple that it can be detached from static basis traversal (which is quite complex), and also makes it easy to report a top-level basis when no evaluation has been done, or where there is no dynamic entity corresponding to the static one (for example, in the case of signatures and functor bodies).

The reporting functions (in `TopLevelReport`) are mutually recursive, and take as argument a function called `render` for performing a dynamic basis traversal at the appropriate leaves of the static basis. For example, the type of `reportStr` is (if we were to label the arguments):

```
signature VAL_PRINT =
   sig
   type id and strid
   type DynamicBasis
   type TypeScheme
   type Val

   val locate:  DynamicBasis * strid list * id -> Val
   val print:  Val * TypeScheme -> string
end
```

Figure 7.4: The signature `VAL_PRINT`

```
{render: StrId list * id * TypeScheme -> string,
 pathR: StrId list,
 str: Str,
 bindings: bool
} -> Report
```

`render` takes a path, an identifier and a type scheme, and prints the dynamic value of that identifier, accessed via the path from the topmost dynamic basis, according to its type as indicated in the type scheme. `pathR` is a (reversed) record of the nested structures traversed to reach the leaf node. The other arguments vary between the different reporting functions, but the argument `bindings`, if present, indicates whether the dynamic value should be retrieved via the `render` function and printed.

`TopLevelReport` makes calls to lower-level functions in other modules. `iterateVE` (in `Environments`) is a generic traverser of value environments; `iterateSE` is the equivalent for structure environments. `ModuleEnvironments` provides traversers for signature and functor environments (which take callback functions for dealing with the individual signatures and functor signatures). Finally, `ValPrint` (whose signature is given in Figure 7.4) provides the ability to traverse a chain of structure identifiers in a dynamic basis yielding a dynamic value, and to print a value according to a type scheme. These are the two operations which will vary in implementation between the interpreter and the compiler.

7.5 Implementation of the Pervasives

Let us refer to the identifiers that are defined in the initial basis of Standard ML as *pervasives*. The pervasives are listed in Appendices C and D of the Definition; they include type constructors (e.g. `int`), value constructors (e.g. `true`) and value variables (e.g. `map`). They also include the basic values, such as `+` and `open_in`.

Some of the pervasives are expressible in terms of ML declarations that use a smaller set of pervasives; this applies to `map`, for example. Such pervasives are defined in a *prelude*

```
fun op = (x: ''a, y: ''a): bool = prim(0, (x, y))
    and abs(x: int): int = prim(1, (x, Abs))
    and floor(x: real): int = prim(2, (x, Floor))
    and real(x: int): real = prim(3, x)
       ...
```

Figure 7.5: Use of `prim` in the prelude

which the Kit reads when building itself.

Other pervasives cannot be dealt with in this way; for instance, `+` is a pervasive value which cannot be expressed in terms of more primitive operations. We say that such pervasives are *primitive*. Pervasives are used by the Kit (for example, `true` is needed in the expansion of derived forms), so pervasives must be accessible to the Kit itself.

In principle, the primitives should include all the basic values (BasVal is defined in [11, Section 6.4]). However, the large number of basic values is a problem in a real implementation. It would be very inconvenient if each basic value were to have its identifier built into the system (separately as an identifier and as a variable), its type built into the elaborator, its value built in as a basic value and its operation built into the APPLY function.

Therefore, the Kit has a special built-in value, called `prim`, which is used to implement most of the basic values in a uniform way. More precisely, most of the basic values (i.e. the members of BasVal) are declared in the prelude, using calls to `prim`. `prim` has type

```
    (int * 'a) -> 'b
```

where the `int` indicates which primitive function is required. It is essentially overloaded, implementing a large number of primitive functions according to the value of its first argument (rather than according to its type, which is the usual discriminant for overloaded operations).

Figure 7.5 gives a small portion of the prelude, showing some uses of `prim`. The second argument to `prim` is either a single value or a tuple expression, depending on the number of arguments the primitive operation takes. Each use of `prim` has to be accompanied by type constraints, since it is a type-unsafe construct, and since the pervasives must be given the types dictated by the Definition. Note also that exception-raising pervasives take exceptions (i.e. values of type `exn`) as extra arguments; the implementation of APPLY can raise these exceptions as required. The equality predicate is implemented using `prim`. (It is prim zero, a fact assumed by the lambda code generator.) The prelude makes all the standard fixity declarations, declares all the pervasive exceptions[2], and declares the types `unit`, `bool` and `list`.

There are a number of things that the prelude cannot implement. These are the following:

- The following types: `int`, `real`, `string`, `exn`, `ref`. The types `int`, `real` and `string`

[2]including `Match`, `Bind` and `Interrupt`, although these are not known by the interpreter at present

need to be built-in because ML has constants of those types. In addition, the type of `prim` contains `int`. `exn` is built-in because of the primitive nature of the exception mechanism. `ref` is built-in because of the special nature of the `ref` constructor, and also because the `ref` type constructor is treated specially when testing types for equality admission.

- The following values: `ref`, `prim`. The need to predefine `prim` should be obvious. `ref` has special static and dynamic properties. Statically, it has type `'a -> 'a ref` in patterns and `'_a -> '_a ref` in expressions. Dynamically, it creates a new address on application, and has a specific semantic rule (rule 114) indicating this.

In addition, some of the non-primitive identifiers are known internally by the system. These are `nil`, `::`, `true`, `false` and `it`. These are needed for derived-form expansion (specifically, `if`/`then`/`else`, list expressions, and top-level expressions). All except `it` are needed for the dynamic evaluation phase as well, since some of the primitive operations return boolean constructors or lists. Note that the types (and identifiers) `bool` and `list` are not known internally. Note also that "`:=`" and "`!`" are non-primitive, and are defined in the prelude with the types required by the Definition.

Perhaps surprisingly, the types `instream` and `outstream` are not primitive. Figure 7.6 shows their definition in the prelude. Their types are introduced using an `abstype` declaration, which makes them suitably opaque. The pervasive operations on streams are provided as primitives (using `prim`) with integers denoting the streams themselves. The interpreter (specifically, the functor `Apply`, using a stream package `IOStreams`) provides the required functionality over streams accessed as integers. This approach frees the rest of the Kit from having to know anything about the types of input and output streams.

The evaluation of the primitives is implemented by the functor `Apply`, whose signature is given in Figure 7.7. The set of basic values BasVal contains a single value, `prim`. `Apply` provides the application function APPLY which takes a BasVal (which must be `prim`) and an argument value (which must be a pair whose first argument is an integer denoting the primitive function required and whose second argument is of the correct form), and returns a result value. `Apply` also provides the equality function as a `prim` (which is used to implement "`=`", and is also needed for pattern matching), and some glue to deal with the `use` function (Section 7.7). The body of `Apply` is fairly straightforward. There are functions for unpacking and packing the primitive types and the important non-primitive types like `bool` and `list`. The function implementing APPLY takes a BasVal (which is ignores, since it can only be `prim`) and a value which is immediately decomposed to yield an integer denoting the primitive operation, and a value denoting the actual arguments to the primitive operation. `Apply` makes use of `IOStreams` to implement streams, and `CoreDynObject.Store` to manipulate references. (Only the `add` and `retrieve` operations are used by APPLY to implement assignment and dereferencing; new references (using `Store.unique`) are created as `ref` is applied, in `EvalDec`.)

`Apply` is also capable of raising exceptions from the primitive functions—it is passed a value `RAISE` of type `CoreDynObject.Val -> 'a` which can be called with exception values.

```
abstype instream = INSTREAM of int
    and outstream = OUTSTREAM of int
with
   val std_in = INSTREAM 0
   val std_out = OUTSTREAM 0

   fun open_in(f: string) =
      INSTREAM(prim(33, (f, Io("Cannot open " ^ f))))

   fun open_out(f: string) =
      OUTSTREAM(prim(34, (f, Io("Cannot open " ^ f))))

   fun input(INSTREAM i, n: int): string = prim(35, (i, n))

   fun lookahead(INSTREAM i): string = prim(36, i)

   fun close_in(INSTREAM i): unit = prim(37, i)

   fun end_of_stream(INSTREAM i): bool = prim(38, i)

   fun output(OUTSTREAM i, str: string): unit =
      prim(39, (i, str, Io "Output stream is closed"))

   fun close_out(OUTSTREAM i): unit = prim(40, i)
end
```

Figure 7.6: Definition of streams from the prelude

```
signature APPLY =
   sig
      type BasVal and Val

      val APPLY: BasVal * Val -> Val
      val equal:  Val * Val -> bool

      val FAIL_USE: unit -> unit
   end
```

Figure 7.7: The signature `APPLY`

7.6 Error Detection and Reporting

As a general rule, whenever a Kit component wants to print some message to the user, it produces a value, called a *report*, which can later be combined with other reports from other parts of the Kit to form a complete message which can then be printed. This approach is very different to eagerly writing something on the screen each time there is some information that the use might want. By delaying printing, we have the possibility of editing reports (or omitting parts of reports altogether) before they are presented to the user.

A report is an abstract representation of a line of text. Reports can be created from strings, appended to each other, indented, decorated with prefix strings, and finally printed. This kind of functionality is needed to print some of the more complex objects of ML such as functor bindings. Rather than print lines of text, a "printing" function will return a report; these reports are accumulated to the top level of the system (functor `Interpreter`) where they are finally printed.

In addition, some printing (or rather, report generation) is deferred. In particular, elaboration errors are not printed (i.e. turned into reports) as soon as they are detected; that would complicate the elaborator, as all the elaboration functions would have to return results containing reports. Instead, elaboration errors cause error info nodes to be planted on the abstract syntax. After elaboration, a separate post-pass runs over the abstract syntax to pick out and to generate a report from any errors which were detected. The interpretation process only continues if there were no errors.

Error traversal is done by `ErrorTraverse`, whose signature is given in Figure 7.8. `ErrorTraverse` examines all info nodes to see if they contain any error info. If so, this is extracted together with the source information (if it is present, which it should be in most cases) and built into a report. The result of each traversal is examined in the top-level loop, in `Interpreter`, where any errors are reported.

Reports are also used for top-level printout of bindings. The functor `TopLevelReport` takes a basis (comprising a static, a dynamic, and an infix basis) and returns a report. `TopLevelReport` is responsible for any indentation and formatting of Modules-level constructs (such as functors, which require some work in order to present them in an acceptable manner), but knows nothing about creating textual representations of Core values. This is done by `ValPrint`; the only assumption made by `TopLevelReport` (perhaps erroneously) is that a value will print into a one-line report.

7.7 File Inclusion and Implementation of `use`

Despite being present in most SML implementations, `use` is not part of the SML Definition. In addition, different implementations implement different functionality for it, and rarely specify its behaviour precisely. Since it is not part of the Definition, it should, strictly speaking, not be implemented in the Kit; however, the Kit's usefulness would be limited without it. Isolated source files could be read using `evalFile`, but it would be difficult to test the Kit on real-world programs since they either contain occurrences of

```
signature ERROR_TRAVERSE =
   sig
      type topdec
      type Report

      datatype result = SUCCESS
                      | FAILURE of Report

      val traverse: topdec -> result
   end
```

Figure 7.8: The signature `ERROR_TRAVERSE`

`use`, or they are built on top of a library or environment (such as the Edinburgh SML Library, or its Make system) which requires `use`.

The Kit does not make `use` pervasive; instead it is defined in a structure called `Kit`, as a call to `prim`. The (informal) semantics are as follows:

- Evaluation of the `use` function applied to a string causes the top-level loop to be immediately invoked on the file, such that the file's bindings (and side-effects) are established in the top-level environment before the call to `use` terminates. Because complete phrases are typechecked before execution, an expression such as

  ```
  (use "file.sml"; x + 1)
  ```

  has the same meaning and value regardless of any binding made to `x` in `file.sml`. The used file is executed in the enclosing basis upto but excluding the phrase containing the call to `use`.

- A compiled-time error (parsing or typechecking) in the used file causes the exception `Use` (in structure `Kit`) to be raised in the calling context. This can be handled in the usual manner.

- An exception which propagates uncaught to the top-level in the used file is passed uncaught into the context calling the `use`, or else reported at top-level.

- Nested occurrences of `use` work as expected.

Therefore, the exception `Use` can be handled in order to deal with compile-time failure for a used file (or any nested file), and runtime exceptions which propagate through a `use` can be handled.

`Interpreter` maintains the top-level basis in a reference variable (all the lower levels are pure in the way they manipulate bases). As each top-level declaration is executed and terminates successfully, its bindings are added to the top-level basis. This means that a file containing an error (compile time or runtime) will cause bindings to be established up to but excluding the offending declaration.

`use` is implemented using source readers (Chapter 3). This means that a call to `use` on a file causes the entire file to be read into memory before its contents are fed to the lexer. In practice, this should be no different behaviourally to the "obvious" scheme where the file is read from sequentially as the lexer is invoked.

The main implementation issue with `use` is that it requires self-reference in the interpreter. `use` is a primitive function which needs to be able to call the entire interpreter on a body of source text, but the interpreter needs to support the entire language, including all the primitive functions such as `use`. The most straightforward implementation technique is to use a reference to hold the `use` function. The reference (of type `(string -> unit) ref`) is provided in functor `KitCompiler`, and is created containing a function of the correct type but no effect. Thus, a function `USE` (of type `string -> unit`) can be passed to the lower levels of the system (the functors `Linking`, `Execution`, `Evaluation`, `EvalDec`, `Apply`). Once the functor heirarchy has been built, the reference variable can be assigned with a function which calls `KitCompiler.interpretFile`.

Exception propagation within `use` is handled as follows. `Apply` has a local exception `Use`, and there is a handler for this at the point in `Apply` where the `use` operation is executed. Should `Use` be raised, the handler translates this failure into a raising of the exception value `Use` which was passed as an argument to `prim` in the prelude. The facility to raise `Use` is exported from `Apply` as the function `FAIL_USE`. This function is passed through `EvalDec` and `EvalTopdec` to `Interpreter`, where the ability to raise `Use` (to signal compilation errors) is needed. For runtime exceptions, `EvalDec` provides a function `RE_RAISE` which the interpreter can use to raise uncaught packets into the surrounding `use` context. (There is no problem with unclosed source files since each used file is read completely into memory and closed before it is presented to the top-level loop.)

Bibliography

[1] Andrew W. Appel. A runtime system. *LISP and Symbolic Computation*, 3:343–380, 1990.

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture.* ACM, Springer-Verlag, Sept 1987.

[4] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A lexical analyzer generator for Standard ML. Online documentation distributed with SML/NJ, 1989.

[5] Andrew W. Appel and David R. Tarditi. ML-Yacc, version 2.1. documentation for release version. Online documentation distributed with SML/NJ, March 1991.

[6] Marianne Baudinet and David MacQueen. Tree pattern matching for ML. Extended abstract, December 1985.

[7] Dave Berry. The edinburgh sml library. Technical Report ECS-LFCS-91-148, Laboratory for Foundations of Computer Science, Department of Computer Science, Edinburgh University, April 1991.

[8] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[9] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[10] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[11] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[12] Didier Rémy. Type inference for records in a natural extension of ML. Technical Report 1431, Inria, Rocquencourt, 1991.

[13] Didier Rémy. Typing record concatenation for free. In *Proc. Principles of Programming Languages(POPL)*, pages 166–176, Jan. 1992.

[14] Donald Sanella and Andrzej Tarlecki. Extended ML: Past, present and future. Technical Report ECS-LFCS-91-138, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

Appendix A

Release Notes

We here describe the latest release of The ML Kit system. Please send comments and bug reports to

Lars Birkedal
DIKU, University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen East
Denmark
Email: birkedal@diku.dk

A.1 Copyright notice

A.2 Getting this release

The ML Kit can be obtained by anonymous ftp from the both the University of Copenhagen, and the University of Edinburgh:

Host	Directory
`ftp.diku.dk`	`diku/users/birkedal`
`ftp.lcfs.ed.ac.uk`	`export/ml/mlkit`

These directories contains the following files

File	Contents
`src.tar.Z`	The src directory containing source code
`doc.tar.Z`	The doc directory containing documentation
`tools.tar.Z`	The tools directory containing some utilities
`examples.tar.Z`	The examples directory containing ... examples
`README`	Summary of release information
`COPYING`	GNU General Public Licence

Here are sample dialogs

```
ftp ftp.diku.dk                        ftp ftp.lfcs.ed.ac.uk
Name: anonymous                        Name: anonymous
Password: <your loginname@host>        Password: <your loginname@host>
ftp> binary                            ftp> binary
ftp> cd diku/users/birkedal            ftp> cd export/ml/mlkit
ftp> get README                        ftp> get README
ftp> get COPYING                       ftp> get COPYING
ftp> get src.tar.Z                     ftp> get src.tar.Z
ftp> get doc.tar.Z                     ftp> get doc.tar.Z
ftp> get tools.tar.Z                   ftp> get tools.tar.Z
ftp> get examples.tar.Z                ftp> get examples.tar.Z
ftp> bye                               ftp> bye
```

After the files are transferred they should be uncompressed and then extracted using tar into a directory called (e.g.) `Kit` For example:

```
mkdir Kit
mv src.tar.Z Kit
cd Kit
zcat src.tar.Z | tar xf -
```

will install the `src` directory (i.e., a directory `src` appears in directory `Kit`).

A.3 Installation

No executable version of the Kit is supplied, i.e., you have to compile the sources yourself. However, this is easily done as described below. The Kit can be compiled using either Standard ML of New Jersey or Poly/ML. However, a stand-alone batch version can only be made using SML/NJ. The tools directory is needed to compile with as described below. Notice, that the executables will be big (around nine megabytes) and it is preferable if you compile using a machine with a lot of memory (it takes around 10 minutes to compile under SML/NJ on a SPARC station with 64 Mb of memory.)

SML/NJ is available from LFCS, as above, but in directory `export/ml/njml`.

Poly/ML is a commercially supported system, available from Abstract Hardware Ltd. (`ahl@ahl.co.uk`)

A.3.1 Compilation under Standard ML of New Jersey

We assume you have a version of SML/NJ with Edinburgh Library loaded (either the full library, or just the Make facility). Then change directory to the `src` directory and set the variable `NJSML` in the `Makefile` to the name of your executable version of SML/NJ. When the `Makefile` has been modified use one of the commands below to build either the interpreter, the compiler or the batch interpreter.

Command	Builds
`make kit-int-NJ`	The interpreter, an executable file `kit` is made
`make kit-comp-NJ`	The compiler, an executable file `kit` is made
`make kit-batch-NJ`	The batch interpreter, an executable file `evalFile` is made

We recommend to build the interpreter and perhaps also the batch interpreter as the compiler is incomplete.

A.3.2 Compilation under Poly/ML

We assume you have a database file with the Edinburgh Library (either the full library, or just the Make facility.) Then simply change directory to the `src` directory and set the variables `POLYDATABASEFILE` and `POLY` in the `Makefile`. When the `Makefile` has been modified use one of the commands below to build either the interpreter or the compiler (no batch system is provided)

Command	Builds
`make kit-int-Poly`	The interpreter, a child database called `kit` is made
`make kit-comp-Poly`	The compiler, a child database called `kit` is made

The database containing the Library is first copied such that this database is not modified by the installation. We recommend to build the interpreter as the compiler is incomplete.

A.4 Non-Standard ML primitives

The Kit interpreter provides a structure `Kit` with the following signature

```
sig
  exception Use
  val use : string -> unit
  val flush_out : outstream -> unit
end
```

The `use` function should be familiar to the reader, `flush_out` flushes its output stream argument and is only provided in order to make it possible to compile the Edinburgh Library using the Kit.

A.5 Compiling the Edinburgh Library using the Kit

Compilation of the Library is done in the following way.

Place yourself in directory `src` and start a kit session; then do the following

```
evalFile "../tools/kit.load"
```

and change directory to the directory containing the portable version of the Library (using `System.cd ".."`). Then do

```
evalFile "build_all.sml"
```

to compile the entire library.[1]

The `kit` session can then be saved as an executable file as described in 1, but beware: the executable is large, around 20 Mb.

[1] The current version of the Library uses Non Standard ML in the file `portable/Make/Make0.sml`. In function `Class` characters are compared using `=`; in Standard ML, function `ord` must be applied before equality is tested

Appendix B

The State of the System

This section lists the unimplemented and nonfunctional features of the Kit, as well as some matters of style and so on which could be improved.

Error conditions in the parsing tools. The shunting algorithm employed by the infix analyser panics on an illegal sequence of operators. It should provide error annotations. An example of offending input is

```
infix @@ val x = @@ 2
```

Lexical errors not handled. The lexer raises an error if it encounters an illegal token (such as a control character, or illegal string escape like `"\ "`). This should be caught and presented as a parsing error.

Overflow in integer constants. There should be a trap for the `Overflow` exception in the lexer, so that large integer constants can be faulted as bad tokens. Currently, the `Overflow` exception propagates.

Missing position information. Source position information is not planted into the abstract syntax created for derived forms. This results in error printing with the source position printed as "`(position unknown)`".

Redefinition of "=". Because the equality predicate is defined in terms of `prim` in the prelude, the Kit must allow "`=`" to appear in a declaration. Since the Definition outlaws this, the Kit should disallow subsequent declarations of "`=`". In the same vein, a qualified identifier containing "`=`" should be disallowed (as it can never be valid).

Numeric record labels. The Definition allows numeric record labels to be of arbitrary size (possibly overflowing an integer), and also syntactically restricts them to not start with a zero. The Kit currently regards record labels as integer constants.

Type path names. Currently, type constructors are printed merely as identifiers. They should be printed as complete paths depending on their accessibility. Inaccessible type constructors should be marked (as "`?.TyCon`" for example). The same should probably be done for exception and data constructors printed at top-level. Similarly, we could insist that values whose data constructors are completely inaccessible be printed opaquely; this would implement `abstype` properly.

Redefinition of `ref`. The Kit currently allows redefinition of `ref` as a data constructor (which is a problem since application of `ref` must be treated specially) and also as a type constructor (which is a problem since equality admission regards `ref` specially).

Treatment of `Match` and `Bind`. Although `Match` and `Bind` are defined in the prelude, they are not raised by inexhaustive pattern matches and bindings.

Modules compiler. There is currently no lambda compiler for the Modules.

Core match compiler. The match compiler for the Core does not deal with exception matches properly. This is a difficult problem, and the current decision tree scheme needs to be rethought to deal with the sequential nature of exception matching.

Parsing cosmetics. There should be a distinct continuation prompt.

Sets instead of lists. Many of the elaboration routines for the Core and Modules use lists in lieu of sets. These should be converted to use the Edinburgh Library's set operations. The main occurrences are those where `ListHacks` and `LIST_HACKS` are currently being used.

Phase problems when changing execution mode. At present, it is possible to run the Kit in elaborate-only mode, and then switch to execute mode with the same accumulated basis. This is insecure, since identifiers will be known statically with the incorrect, or no, runtime binding.

Occurrences of `op`. In `Infixing`, there should be a cosmetic check for occurrences of `op` in datatype and exception bindings. According to the Definition, `op` is required for identifiers with infix status, although the parser does not need this annotation.

Bad functor argument reporting. The declaration "`functor F() = struct end`" gets turned into a report with what looks like the first line missing.