Standard ML Reference Manual
(PRELIMINARY)

December 23, 1992

Abstract

This manual is a major revision by Andrew W. Appel and David B. MacQueen of the *Standard ML* reference manual (ECS-LFCS-86-2). That document is divided into three parts: the Core language description by Robin Milner, the standard I/O library by Robert W. Harper, and the Module system by David B. MacQueen. This new manual reflects a slightly evolved language, and describes the library functions (initial environment) built into the Standard ML of New Jersey system, developed at AT&T Bell Laboratories and Princeton University.

At present the manual is in a very rough form, and should be considered a preliminary draft. This version is distributed primarily as documentation of the mid-1989 distribution of the *Standard ML of New Jersey* compiler.

Chapter 1

Introduction

This is a preliminary draft of a reference manual for Standard ML of New Jersey, an implementation of the Standard ML language. Our implementation adheres closely to the Standard ML language definition [1] except that the library of built-in functions is larger and better organized.

The document at present has many gaps. It should be treated as a guide to the 1989 pre-release, rather than as a complete reference manual.

[1] see *The Definition of Standard ML, version 3*, by Robert Harper, Robin Milner, and Mads Tofte, technical report ECS-LFCS-89-81, Dept. of Computer Science, Univ. of Edinburgh, Edinburgh EH9 3JZ Scotland.

Chapter 2

Lexical analysis

2.1 Reserved words

The following are reserved words. They may not be used as identifiers. In this document the alphabetic reserved words are always shown in boldface.

```
abstraction abstype and andalso as case datatype else
end exception do fn fun functor handle if in infix
infixr let local nonfix of op open overload raise rec
sharing sig signature struct structure then type val
while with withtype orelse

{ } [ ] , ; ( ) -> * | : ... = => # _
```

2.2 Special constants

An integer constant is any non-empty sequence of digits, possibly preceded by a negation symbol (~).

A real constant is an integer constant, possibly followed by a point (.) and one or more digits, possibly followed by an exponent symbol(E) and an integer constant; at least one of the optional parts must occur, hence no integer constant is a real constant. Examples: `0.7` , `~3.32E5` , `3E~7` . Non-examples: `23` , `.3` , `4.E5` , `1E2.0` .

A string constant is a sequence, between quotes ("), of zero or more printable characters, spaces, or escape sequences. Each escape sequence is introduced by the escape character \, and stands for a character sequence. The allowed escape sequences are as follows (all other uses of \ being incorrect):

`\n`	A single character interpreted by the system as end-of-line.
`\t`	Tab.
`\^c`	The control character c, for any appropriate c.
`\ddd`	The single character with ASCII code ddd (3 decimal digits).
`\"`	The double-quote character (").
`\\`	The backslash character (\).
`\f___f\`	This sequence is ignored, where f___f stands for a sequence of one or more formatting characters (a subset of the non-printable characters including at least space, tab, newline, formfeed). This allows one to write long strings on more than one line, by writing \ at the end of one line and at the start of the next.

2.3 Identifiers

An identifier is either *alphanumeric*: any sequence of letters, digits, primes (`'`), and underbars (`_`) starting with a letter or a prime, or *symbolic*: any sequence of the following symbols

```
! % & $ + - / : < = > ? @ \ ~ \^ | # * `
```

In either case, however, reserved words are excluded. This means that for example `_` and `|` are not identifiers, but `also_ran` and `|=|` are identifiers.

Identifiers are used to stand for 9 different classes of objects, which occupy 6 different name spaces, as follows:

1. value variables (*var*), value constructors (*con*), exception constructors (*exncon*)
2. type variables (*tyvar*)
3. type constructors (*tycon*)
4. record labels (*lab*)
5. structures (*str*), functors (*fct*)
6. signatures (*sgn*)

Thus, an identifier could not in the same scope stand for both a value variable and a constructor, but an identifier can be bound simultaneously to a type constructor and a signature.

To remove some ambiguity, it is recommended that constructors start with an uppercase letter, and variables start with a lowercase letter; but this is a convention, not an enforced rule (it is confounded, for example, by symbolic identifiers).

A type variable (*tyvar*) may be any alphanumeric identifier starting with a prime. The other eight classes (*var, con, tycon, ...*) are represented by identifiers not starting with a prime. The class lab is also extended to include the numeric labels 1, 2, 3, ...

Type variables are therefore disjoint from the other classes. Otherwise, the class of an occurrence of an identifier is determined from context.

Spaces or parentheses are sometimes needed to separate symbolic identifiers and reserved words. Two examples are

`a:= !b`	or	`a:=(!b)`	but not	`a:=!b`
`~ :int->int`	or	`(~):int->int`	but not	`~:int->int`

These punctuation characters cannot be constituents of identifiers and therefore never need spaces around them:

```
" ( ) , . ; [ ] { }
```

2.4 Comments

A comment is a character sequence (outside of a string) within comment brackets (* *) in which comment brackets are properly nested.

2.5 The bare syntax

The Standard ML bare language is obtained by stripping the full language of any *derived* forms (those that may be defined in terms of other constructs in the language), and of any constructs related to the module system. The bare language will be explained in Chapters 3 and 4, and successive chapters describe augmentations of it that yield the full language.

Figure 2.5 shows the syntax of the bare language. The notation

phrase x $\underline{\ k\ }$ x phrase

indicates the repetition of the *phrase* at least k times, separated by the punctuation character x.

Figure 2.5: Syntax of the Bare Language

lab → INT
  id

const → INT
  REAL
  STRING

exp → id
  const
  { lab=exp , $\underline{\;0\;}$, lab=exp }
  let dec **in** exp **end**
  exp exp
  exp : ty
  fn match
  raise exp
  exp **handle** match

match → pat => exp | $\underline{\;1\;}$ | pat => exp

apat → id
  const
  _
  (pat)
  { lab=pat , $\underline{\;0\;}$, lab=pat }
  { lab=pat , $\underline{\;0\;}$, lab=pat , ... }

pat → apat
  id apat
  pat : ty
  id constraint **as** pat

ty → **tyvar**
  { lab : ty , $\underline{\;0\;}$, lab : ty }
  (ty)
  (ty , $\underline{\;2\;}$, ty) id
  ty id
  id
  ty -> ty

vb → pat = exp
  vb **and** $\underline{\;1\;}$ **and** vb

constraint → *empty*
  : ty

rvb → id constraint = **fn** match
  rvb **and** $\underline{\;1\;}$ **and** rvb

tb → tyvars id = ty
  tb **and** $\underline{\;1\;}$ **and** tb

tyvars → *empty*
  tyvar
  (**tyvar** , $\underline{\;2\;}$, **tyvar**)

db → tyvars id = constr | $\underline{\;1\;}$ | constr
  db **and** $\underline{\;1\;}$ **and** db

constr → id
  id **of** ty

eb → id
  id **of** ty
  id = id
  eb **and** $\underline{\;1\;}$ **and** eb

dec → **val** vb
  val **rec** rvb
  type tb
  datatype db
  exception eb
  local dec **in** dec **end**
  dec ;
  dec dec

Chapter 3

Evaluation

3.1 Environments and Values

Evaluation of phrases takes place in the presence of an *environment* and a *store*. An *value environment* E maps identifiers to values, value constructors, and exception constructors. A *store* S maps references to values; references are themselves values. (Type environments TE, are ignored here since they are relevant only to type-checking and compilation, not to evaluation. Other environments having to do with the module system are also ignored here.)

A value v is either a constant (a nullary constructor), a construction (a constructor applied to a value), a record, a reference, or a function value. A record value is a set of label-value pairs, written { label = value , $\underline{\ 0\ }$, label = value }, in which the labels are distinct; note that the order of components is immaterial. Labels may be identifiers or integer numerals; both kinds of labels may appear in the same record value. A function value is a partial function which, given a value, may return a value or may raise an exception; it may also change the store as a side-effect.

An exception e, associated to an exception name *exn* in a value environment, is a special kind of constructor. Exception constructors may be nullary or value-carrying, just as may ordinary constructors. Nullary exception constructors, and constructed exceptions (exception constructors applied to values), are ordinary values of type *exn*.

Evaluation of a phrase returns a *result*—a value v, a value-environment E, or a store S as follows:

Phrase	Value
Expression	v and S, or raise(v) and S
Value binding	E and S, or raise(v) and S
Type binding	*no effect on E or S*
Datatype binding	E
Exception binding	E
Declaration	E and S, or raise(v) and S

The remainder of this chapter describes the semantics of various phrases in terms of values v and environments E.

The semantics of stores (S) are discussed in Chapter 10; for the remainder of this chapter, the effect of various phrases on stores will be ignored. The semantics of exceptions is discussed in Chapter 9.

For every phrase except a `handle` expression, whenever its evaluation demands the evaluation of an immediate subphrase which returns a raised exception *raise(v)* as a result, no further evaluation of subphrases occurs and *raise(v)* is also the result of the phrase. This rule should be remembered while reading the evaluation rules below.

In presenting the rules, explicit type constraints (:ty) have been ignored since they have no effect on evaluation.

3.2 Environment manipulation

We may write $\{i_1 \mapsto v_1, ..., i_n \mapsto v_n\}$ for a value environment E (the identifiers i_k being distinct). Then $E(i_k)$ denotes v_k, $\{\}$ is the empty value environment, and $E + E'$ means the environment in which the associations of E' supersede those of E.

3.3 Matching patterns

Patterns serve in ML both as formal parameters of functions, and as indices in case statements. When a function or a case statement is applied to a particular value, one of its patterns may match the value.

A nullary constructor (like `nil`) used as a pattern will match only the corresponding nullary constructor value. A value-carrying constructor c, applied to a pattern p_1, makes a pattern $c(p_1)$; this constructed pattern will match a value $c(v_1)$ built with the same value-carrying constructor, and only if the pattern p_1 matches the value v_1.

A record pattern `{` lab_1 `=` pat_1 `,` __ `,` lab_n `=` pat_n `}` matches a record value whose labels are the same, and only if each pattern matches the corresponding value. If the ellipsis (`...`) is used at the end of a record pattern, then it will match a record value with *at least* the labels specified in the pattern.

An identifier (i.e. a variable) used as a pattern will match any value; when this happens, the variable will also be bound to that value throughout the variable's scope. When a variable is a component of a record or constructor pattern, then it is bound to a particular component of the record value or constructed value.

An underscore will match any value.

Sometimes it is desired to bind a variable to a value only if the value matches a particular pattern. The pattern *id* `as` *pat* binds the value to the variable *id*, but only if the *pat* matches.

Type constraints may be applied to patterns. *pat*: *ty* matches the same values that *pat* does, but the compiler will guarantee that the pattern will only be applied to values of type *ty*.

A more formal description

The matching of a pattern to a value v either *fails* or yields a value environment. Failure is distinct from raising an exception, but an exception will be raised when all patterns fail in applying a match to a value (see Sections 3.5, 3.6, and 9.2). In the following rules, if any component pattern fails to match then the whole pattern fails to match.

The following is the effect of matching a pattern to a value v in the environment E, for each of the kinds of pattern (with failure if any condition is not satisfied):

_	*(underscore)* the empty value environment $\{\}$ is returned
id	if $E(id)$ is not a constructor, then the value environment $\{id \mapsto v\}$ is returned
id	if $E(id)$ is a nullary constructor, then if $E(id) = v$, then $\{\}$ is returned, else failure
id pat	if $E(id)$ is a value-carrying constructor c, and $v = cv'$, then pat is matched to v', else failure.
id as pat	pat is matched to v returning E; then $\{id \mapsto v\} + E$ is returned.
{ lab_1 = pat_1 , ___ , lab_n = pat_n }	if $v = \{lab_1 = v_1, ..., lab_n = v_n\}$ then pat_i is matched to v_i returning E_i, for each i; then $E_1 + ... + E_n$ is returned.
{ lab_1 = pat_1 , ___ , lab_n = pat_n , ... }	if $v = \{lab'_1 = v_1, ..., lab'_m = v_m\}$ then if the lab_i are a subset of the lab'_j then pat_i is matched to v_j returning E_i, for each i, j such that $lab_i = lab'_j$; then $E_1 + ... + E_n$ is returned.

No pattern may contain the same variable twice. No record pattern, record expression, or record type may use the same label twice.

3.4 Applying a match

Assume environment E. Applying a match pat_1 => exp_1 | ... | pat_n => exp_n to value v returns a value or raises an exception as follows. Each pat_i is matched to v in turn, from left to right, until one succeeds returning E_i; then exp_i is evaluated in $E + E_i$. If none succeeds, then an exception is raised, depending on the syntactic context in which the match occurs (see Sections 3.5, 3.6, and 9.2). If a match contains a redundant pattern (where any value that could satisfy it will be matched by a previous pattern in the match), the compiler will issue a warning message. If the match (except those that form exception-handlers) is not exhaustive (some value matches none of the patterns) then the compiler will issue a warning message.

Thus, for each E, a match denotes a function value.

3.5 Evaluation of expressions

Evaluating an expression in the environment E returns a value (or raises an exception) as follows, in each of the cases for exp:

id — the value $E(id)$ is returned; id may be a variable-id or a constructor-id

const — the value denoted by the constant (an integer, real, or string literal) is returned.

exp$_1$exp$_2$ — **exp**$_1$ is evaluated, returning function f; then exp_2 is evaluated, returning value v; then $f(v)$ is returned.

{ lab_1 = exp_1 , ___ , lab_n = exp_n } — the exp_i are evaluated in sequence, from left to right, returning v_i respectively; then the record $\{lab_1 = v_1, ..., lab_n = v_n\}$ is returned.

`raise` **exp** — exp is evaluated, returning v; then the exception-value v is raised.

exp `handle` **match** — exp is evaluated; if exp returns a value v, then v is returned. If exp raises an exception e then the match is applied to e. If the match fails, then e is raised (as the value of the `handle` expression). If the match succeeds, then the resulting value is returned.

`let` **dec** `in` **exp** `end` — dec is evaluated, returning E'; then exp is evaluated in the environment $E + E'$.

`fn` **match** — f is returned, where f is the function of v gained by applying match to v in the environment E, and such that if the match fails, an exception `Match` will be raised. But matches that may fail are to be detected by the compiler and flagged with a warning; see Section 3.4.

3.6 Evaluation of value bindings

Evaluating a value binding in environment E returns a value environment E' or raises an exception as follows:

pat = **exp** — exp is evaluated in E, returning value v; then pat is matched to v; if this returns E' then E' is returned. If the pattern fails to match then the exception `Bind` will be raised. But bindings that may fail are to be detected by the compiler and flagged with a warning; see Section 3.6.

vb$_1$ `and` __ `and` **vb**$_n$ — The value bindings **vb**$_1$ through **vb**$_n$ are evaluated in E from left to right, returning $E_1, ...E_n$; then $E_1 + ... + E_n$ is returned.

`rec` **vb** — **vb** is evaluated in E', returning E'', where $E' = E + E''$. Because the values bound by `rec` **vb** must be function values, E' is well defined by "tying knots" (Landin).

No binding may bind the same identifier twice.

For each value binding "pat = exp" the compiler will issue a warning message if *either* pat is not exhaustive *or* pat contains no variable. This will (on both counts) detect a mistaken declaration like `val nil = f(x)` in which the user expects to declare a new variable nil (whereas the language dictates that `nil` here is a constant pattern, so no variable gets declared). However, these warnings need not be given at top level in the interactive system.

3.7 Evaluation of type and datatype bindings

The value environment E does not affect the evaluation of type bindings or datatype bindings (TE affects their type-checking and compilation). Evaluation of a type binding just returns the empty value environment $\{\}$; the purpose of type bindings is merely to provide an abbreviation for a compound type.

Evaluation of a datatype binding **db** returns a value environment E' (it cannot raise an exception) as follows:

tyvars id = **constr**$_1$ | __ | **constr**$_n$ — New constructors **con**$_1, ..., $**con**$_n$ are created. The value environment $E' = \{id_1 \mapsto$

$v_1, ..., id_n \mapsto v_n\}$ is returned, where v_i is either the constant value $\mathbf{con}_i$ (if $\mathbf{constr}_i$ is just $\mathbf{id}_i$), or else the function which maps x to $\mathbf{con}_i(x)$ (if $\mathbf{constr}_i$ is $\mathbf{id}_i$ `of` **ty**).

$\mathbf{db}_1$ `and` ___ `and` $\mathbf{db}_n$ — The bindings $\mathbf{db}_1, ..., \mathbf{db}_n$ are evaluated from left to right, returning $E_1, ..., E_n$; then $E_1 + ... + E_n$ is returned.

In the left hand side "tyvars id" of a type of datatype binding, no type-variable may appear twice in "tyvars." The right hand side may contain only the type variables mentioned on the left. Within the scope of the declaration of "id," any occurrence of the type constructor "id" must be accompanied by as many type arguments as indicated by the (possibly empty) tyvar sequence "tyvars" in the declaration.

No binding may bind the same identifier twice.

3.8 Evaluation of exception bindings

The evaluation of an exception binding in an environment E returns an environment E' as follows:

id — A new exception constructor **con** is generated, and the environment $\{\mathbf{id} \mapsto \mathbf{con}\}$ is returned.

id `of` **ty** — A new exception constructor **con** is generated, and the environment $\{\mathbf{id} \mapsto v\}$ is returned, where v is the function of x that returns $\mathbf{con}(x)$.

$\mathbf{id}_1$ `=` $\mathbf{id}_2$ — The environment $\{\mathbf{id}_1 \mapsto E(\mathbf{id}_2)\}$ is returned; that is, $\mathbf{id}_1$ is bound to the same exception constructor as $\mathbf{id}_2$.

$\mathbf{eb}_1$ `and` ___ `and` $\mathbf{eb}_n$ — The bindings $\mathbf{eb}_1, ..., \mathbf{eb}_n$ are evaluated from left to right, returning $E_1, ..., E_n$; then $E_1 + ... + E_n$ is returned.

No binding may bind the same identifier twice.

3.9 Evaluation of declarations

Evaluating a declaration in a value environment E returns a value environment E' or raises an exception as follows (as usual in this chapter, the effect on type environments is ignored):

`val` vb — The value binding vb is evaluated, returning E'; then E' is returned.

`type` tb — The empty environment $\{\}$ is returned.

`datatype` db — db is evaluated, returning E', which is returned.

`exception` eb — eb is evaluated, returning E', which is returned.

`local` $\mathbf{dec}_1$ `in` $\mathbf{dec}_2$ `end` — dec_1 is evaluated in E, returning E_1; then dec_2 is evaluated in $E+E_1$, returning E_2; then $E+E_2$ is returned.

$\mathbf{dec}_1$ $\mathbf{dec}_2$ — dec_1 is evaluated in E, returning E_1; then dec_2 is evaluated in $E + E_1$, returning E_2; then $E_1 + E_2$ is returned.

dec ; — has the same effect as **dec**.

3.10 Evaluation of a program

The evaluation of a program $\mathbf{dec}_1$; __ ; $\mathbf{dec}_n$ takes place in the initial presence of the standard top-level environment E_0 containing all the standard bindings (see Appendix B). For $i > 0$ the top-level environment E_i, present after the evaluation of $\mathbf{dec}_i$ in the program, is defined recursively as follows: $\mathbf{dec}_i$ is evaluated in E_{i-1} returning environment E'_i, and then $E_i = E_{i-1} + E'_i$.

Chapter 4

The type system

Type checking is much the same as in the previous system. This chapter has not yet been written.

Chapter 5

Directives

Directives are included in ML as (syntactically) a subclass of declarations. They possess scope, as do all declarations. The directives

`infix` d $\mathbf{id}_1$ __ $\mathbf{id}_n$

`infixr` d $\mathbf{id}_1$ __ $\mathbf{id}_n$

introduce infix status for the identifiers $\mathbf{id}_1$ through $\mathbf{id}_n$. The digit d (optional, with a default of 0) determines the precedence, and an infixed identifier associates to the left if introduced by `infix`, and to the right if introduced by `infixr`. Different infixed operators of equal precedence associate to the left. As indicated in Appendix A, the precedence of infixed application is just weaker than that of application.

The directive

`nonfix` $\mathbf{id}_1$ __ $\mathbf{id}_n$

cancels infix status for the named identifiers.

While an identifier has infix status, each occurrence of it (as a value variable or as a constructor) must be infixed or else preceded by `op`. Note that this includes occurrences of the identifier within patterns, even binding occurrences of variables.

Several standard functions and constructors have infix status (see Appendix A) with precedence; these are all left associative except "`::`".

Chapter 6

Standard bindings

ML provides the record type constructor $\{\mathbf{lab}_1 : \mathbf{ty}_1, \_\_, \mathbf{lab}_n : \mathbf{ty}_n\}$ for any set $\{\mathbf{lab}_i\}$ of labels and corresponding set $\{\mathbf{ty}_i\}$ of types. The language also provides the infixed function-type constructor `->`. Otherwise, type constructors are postfixed. The following are standard:

Type *constants* **(nullary constructors):** unit, bool, exn, int, real, string, instream, outstream

Unary type constructors: list, ref

The constructors unit, bool, and list are fully defined by the following assumed declaration

```
infixr 5 ::
type unit = {}
datatype    bool = true | false
datatype  'a list = nil |  :: of {1 : 'a, 2 : 'a list}
```

The word "unit" is chosen since the type contains just one value "`{}`", the empty record. This is why it is preferred to the word "void" of Algol-68.

The type constants `int`, `real`, and `string` are equipped with special constants as described in section 2.3. The type constructor `ref` is for constructing reference types; see Chapter 10. The type constant `exn` is the type of all exceptions, and is a datatype containing an unbounded number of constructors generated by `exception` bindings (see Chapter 9).

The initial top-level environment is comprised of a set of standard bindings. The initial environment is much more extensive than the environment described in *The Definition of Standard ML*, and is almost a proper superset. The differences are:

- All values, types, datatypes, etc. are grouped into structures; these structures are opened in the initial environment so that the names can be used without a structure-name qualification.

- There are many additional initial bindings, described in Appendix B.

- The functions `input` and `output` are curried in Standard ML of New Jersey, e.g. `input: instream->int->string` instead of `input: instream*int->string`.

- The integer functions `+`, `-`, `div`, `mod`, `*`, `~`, `abs` all raise `Overflow` if the result is out-of-range, rather than `Sum`, `Diff`, `Div`, `Mod`, `Prod`, `Neg`, `Abs`, respectively.

- The integer `div` function rounds toward zero, and `x mod y` is defined as `x-y*(x div y)`.

Standard ML of New Jersey is distributed with a structure `Standard` that may be loaded into the initial environment to simulate environment described in the *Definition*.

Chapter 7

Derived forms

ML is equipped with a number of *derived forms*, which in no way add to the power of the language, as each is expressible in terms of the more primitive constructs.

The n-tuple type `(`ty_1`*`ty_2`*` $\cdots$ `*`ty_n`)` for $n \geq 2$ is an abbreviation for the record type with numeric labels `{1:`ty_1`,2:`ty_2`,` $\cdots$ `,`n`:`ty_n`}`. Similarly the n-tuple expression `(`exp_1`,`exp_2`,` $\cdots$ `,`exp_n`)` is an abbreviation for the record expression verb"1=" exp_1`,2=`exp_2`,` $\cdots$ `,`n`=`exp_n`}`, and the n-tuple pattern `(`pat_1`,`pat_2`,` $\cdots$ `,`pat_n`)` is an abbrevation for the record pattern verb"1=" pat_1`,2=`pat_2`,` $\cdots$ `,`n`=`pat_n`}`.

The "empty record" `{}` can also be written as `()`. This value is conventionally returned from functions that have side-effects but don't return an interesting value. The type of empty records is named `unit` (because the type only contains one value).

The expression `#`*lab* for any label (symbolic or numeric) is a selector function that extracts the named field from a record.

The `case` *exp* `of` *match* expression is completely equivalent to a `(fn` *match*`)` expression applied to the *exp*.

The `if-then-else` expression has conventional semantics: it evaluates a boolean condition, then evaluates either the `then` expression or the `else` expression; this behavior can be defined in terms of a `case` with patterns `true` and `false`.

The `orelse` and `andalso` boolean operators evaluate their right-hand expressions only if the left-hand expressions are insufficient to determine the answer (i.e. `false` for `orelse` and `true` for `andalso`). Their syntactic precedence is weaker than all other infix operators, and `orelse` binds weaker than `andalso`.

Several expressions may be separated by semicolons, and the whole enclosed in parentheses; all the expressions will be evaluated and the result of the whole will is the result of the last expression. When such an *expression sequence* is the entire body of a `let` expression, the parentheses may be omitted.

The expression `while` exp_1 `do` exp_2 repeatedly evaluates exp_1 followed by

exp_2 until exp_1 evaluates to `false` (just as in Pascal); the `while` expression can be expressed in terms of a recursive function.

The expression `[`exp_1`,`exp_2`,`$\cdots$`,`exp_n`]` is an abbreviation for the list exp_1`::`exp_2`::`$\cdots$`::`exp_n`::nil`, and similarly the pattern `[`pat_1`,`pat_2`,`$\cdots$`,`pat_n`]` is an abbreviation for the list pattern pat_1`::`pat_2`::`$\cdots$`::`pat_n`::nil`. In both patterns and expressions, `[]` is an abbrevation for `nil`.

In a record pattern (but not in a record expression), an element *id*`=`*id*, where the pattern is a variable with the same identifier as the label, can be abbreviated as just *id*. Similarly, *id*`=~`*id* verb"as" *pat* can be abbreviated as "*id* verb"as" *pat*, *id*`=~`*id* verb":" *ty* can be abbreviated "*id* verb":" *ty*.

Recursive clausal function definitions can be defined conviently with the `fun` keyword. The declaration

```
fun f pat1a pat2a pat3a = exp1
  | f pat1b pat2b pat3b = exp2
  | f pat1c pat2c pat3c = exp2
```

is an abbreviation for the curried function

```
val rec f = fn pat1 => fn pat2 => fn pat3 =>
              case (pat1,pat2,pat3)
               of (pat1a,pat2a,pat3a) => exp1
                | (pat1b,pat2b,pat3b) => exp1
                | (pat1c,pat2c,pat3c) => exp1
```

The patterns must all be atomic patterns (including, of course, arbitrary patterns enclosed in parentheses) to avoid syntactic ambiguity.

A special form of `fun` declaration is permitted for infixed function identifiers, of which an example is shown here:

```
fun a + b = b - ~a
```

The derived forms are summarized in the tables below.

7.1 Expressions and patterns

<table>
<tr><th>Derived Form</th><th>Equivalent Form</th></tr>
<tr><td>Types:</td><td></td></tr>
<tr><td>ty$_1$ * $\underline{\;2\;}$ * ty$_n$</td><td>{ 1 : ty$_1$, $\underline{\;2\;}$, n : ty$_n$ }</td></tr>
<tr><td>Expressions:</td><td></td></tr>
<tr><td>()</td><td>{ }</td></tr>
<tr><td>(exp$_1$, $\underline{\;2\;}$, exp$_n$)</td><td>{ 1 = exp$_1$, $\underline{\;2\;}$, n = exp$_n$ }</td></tr>
<tr><td>case exp of match</td><td>(fn match) (exp)</td></tr>
<tr><td># lab</td><td>fn { lab = x , ...} => x</td></tr>
<tr><td>if exp then exp$_1$ else exp$_2$</td><td>case exp of true => exp$_1$
| false => exp$_2$</td></tr>
<tr><td>exp$_1$ orelse exp$_2$</td><td>if exp$_1$ then true else exp$_2$</td></tr>
<tr><td>exp$_1$ andalso exp$_2$</td><td>if exp$_1$ then exp$_2$ else false</td></tr>
<tr><td>(exp$_1$; $\underline{\;1\;}$; exp$_n$; exp)</td><td>case exp$_1$ of _ => $\underline{\;\;}$ =>
case exp$_n$ of _ => exp</td></tr>
<tr><td>let dec in exp$_1$; $\underline{\;1\;}$; exp$_n$ end</td><td>let dec in (exp$_1$; $\underline{\;1\;}$; exp$_n$) end</td></tr>
<tr><td>while exp$_1$ do exp$_2$</td><td>let val rec f = fn () =>
if exp$_1$ then (exp$_2$; f()) else ()
in f() end</td></tr>
<tr><td>[exp$_1$, $\underline{\;0\;}$, exp$_n$]</td><td>exp$_1$ :: $\underline{\;0\;}$ :: exp$_n$ :: nil</td></tr>
<tr><th>Derived Form</th><th>Equivalent Form</th></tr>
<tr><td>Patterns:</td><td></td></tr>
<tr><td>()</td><td>{ } (no space between "()")</td></tr>
<tr><td>(pat$_1$, $\underline{\;2\;}$, pat$_n$)</td><td>{ 1 = pat$_1$, $\underline{\;2\;}$, n = pat$_n$ }</td></tr>
<tr><td>[pat$_1$, $\underline{\;0\;}$, pat$_n$]</td><td>pat$_1$ :: $\underline{\;0\;}$ :: pat$_n$ :: nil</td></tr>
<tr><td>{ $\underline{\;\;}$, id , $\underline{\;\;}$ }</td><td>{ $\underline{\;\;}$, id = id , $\underline{\;\;}$ }</td></tr>
<tr><td>{ $\underline{\;\;}$, id as pat, $\underline{\;\;}$ }</td><td>{ $\underline{\;\;}$, id = id as pat, $\underline{\;\;}$ }</td></tr>
<tr><td>{ $\underline{\;\;}$, id : ty , $\underline{\;\;}$ }</td><td>{ $\underline{\;\;}$, id = id : ty , $\underline{\;\;}$ }</td></tr>
</table>

Each derived form is identical semantically to its "equivalent form." The type-checking of each derived form is also defined by that of its equivalent form. The precedence among all primitive and derived forms is shown in Appendix A.

The derived type ty$_1$ * $\underline{\;2\;}$ * ty$_n$ is called an (n–)tuple type, and the values of this type are called (n–)tuples.

The final derived pattern allows a label and its associated value to be elided in a record pattern, when they are the same identifier.

7.2 Bindings and declarations

A syntax class **fb** of function bindings is used as a convient form of value binding for (possibly recursive) function declarations. The equivalent form of each function binding is an ordinary value binding. These new function bindings must be declared by `fun`, not by `val`; however, functions may still be declared using `val` or `val rec` along with `fn` expressions.

Derived Form	**Equivalent Form**
Function bindings fb:	
	id = `fn` x_1 `=>` $\underline{\ 1\ }$ `=>` `fn` x_n `=>`
	`case` (x_1, $\underline{\quad}$, x_n)
id apat_{11} $\underline{\ 1\ }$ apat_{1n} cst = exp_1	`of` (apat_{11} , $\underline{\ 1\ }$, apat_{1n} `=>` exp_1 cst
$\mid$ $\underline{\quad}$	$\mid$ $\underline{\quad}$
$\mid$ id apat_{m1} $\underline{\ 1\ }$ apat_{mn} cst = exp_m	$\mid$ (apat_{m1} , $\underline{\ 1\ }$, apat_{mn} `=>` exp_m cst
fb_1 and $\underline{\ 1\ }$ and fb_n	vb_1 and $\underline{\ 1\ }$ and vb_n
	(where vb_i *is the equivalent of* fb_i*)*
Declarations:	
`fun` fb	`val rec` vb
	(where vb *is the equivalent of* fb*)*
exp	`val it =` exp *(only at top level)*

In the table above, "cst" stands for an optional type constraint—a colon followed by a type expression. The last derived declaration (using "it") is only allowed at top-level, for treating top-level expressions as degenerate declarations; "it" is just a normal value variable.

Chapter 8

Equality

The equality function `op = : ''a * ''a -> bool` is available at all types `''a` except function types, abstract types, and the types constructed from them. In fact, type variables that begin with two primes are special: they stand only for types that admit equality.

Two values are tested for equality as follows, depending on the kind of value:

Primitive types like integers, reals, and strings have equality functions with the conventional behavior.

Function types cannot be compared ("do not admit equality").

Reference types: On references, equality means identity; a reference is equal to itself and to no other references, regardless of similar contents.

Record types may be compared if all their components admit equality.

Datatypes may be compared for equality if all of their constructed types admit equality.

Opaque types from functor parameters and abstractions do not admit equality unless the `eqtype` keyword is used (instead of the `type` keyword) in the signature defining them.

The function `op <> : ''a * ''a -> bool` is the inequality function; it is applicable to any equality type.

The comparison functions `>`, verb"¡", `<=`, and `>=` do not have this behavior; they are overloaded just for the types `int`, `real`, and `string`.

Chapter 9

Exceptions

The rough and ready rule for understanding how exceptions are handled is as follows. If an exception is raised by a `raise` expression

`raise` E `(` *exp* `)`

which lies in the textual scope of a declaration of the exception constructor E, then it may be handled by a handling rule

`handle` E `(` *pat* `) =>` *exp'*

but only if this handler is in the textual scope of the same declaration.

Any exception, regardless of scope, is handled by a wildcard or variable pattern, as

`handle _ =>` *exp'*

This rule is perfectly adequate for exceptions declared at top level; some examples in Section 9.3 below illustrate what may occur in other cases.

9.1 An example

To illustrate the generality of exception handling, suppose we have declared some exceptions as follows:

```
exception    Oddlist of int    and    Oddstring of string
```

and that a certain expression exp:int may raise either of these exceptions and also runs the risk of dividing by zero. The handler in the following `handle` expression would deal with these expressions:

```
exp handle Oddlist []  => 0
         | Oddlist [x] => 2*x
         | Oddlist (x::y::_) => x div y
         | Oddstring "" => 0
         | Oddstring s => size(s)-1
         | Div => 10000
```

Note that the whole expression is well-typed because in each handling rule the type of the match-pattern is `exn`, and because the result type of the match is `int`, the same as the type of exp.

Note also that the last handling rule will handle `Div` exceptions raised by exp, but will not handle the `Div` exception that may be raised by `x div y` in the third handling rule. Finally, note that a universal handling rule

```
         | _ => 50000
```

at the end would deal with all other exceptions raised by exp.

9.2 Exception constructors

For an exception constructor E, the expression

E (*exp*)

evaluates the expression *exp*, producing value v, and then applies the constructor E to it, yielding the value $E(v)$, whose type is `exn`.

The `raise` keyword may be applied to any expression of type `exn`, and has the effect of "raising" that exception value. The innermost (dynamically) enclosing expression $e = e_1$ `handle` m_1 is found; all further evaluation of the expression e_1 (and its subphrases) is aborted; and the match m_1 is applied to the exception value, yielding the result of the expression e.

If the match in a handler fails, then the exception value is re-raised, and another enclosing handler is found.

Exception constructors may be nullary (have no associated value), in which case the *exp* and *pat* in the previous discussion are omitted.

Exceptions may be constructed independently of raising them:

```
exception A of int
val e = A 6
val x = raise e
```

Handlers may be abstracted from the `handle` keyword:

```
val h = fn E 0 => "zero"
         | E _ => "nonzero"
         | v => raise v

f(x)  handle  e => h e
```

Note that it is advisable in this case to have a default clause in the function `h`, since the default for a `handle` match (re-raising the exception) is different from the default for a `fn` or `case` match (raising the `Match` exception).

The ordinary wildcard pattern _ will handle any exception when it is used in a pattern, as will any pattern consisting solely of a variable. These should be used with some care, bearing in mind that they will even handle interrupts.

Nullary exception names, when misspelled, appear to the compiler to be variables; these will then match any exception. For this reason we recommend the convention that exception names (and other constructors) be written beginning with an uppercase character, and variables be written beginning with a lowercase character. The compiler may remind the programmer of this convention when it is violated.

9.3 Some pathological examples

We now consider some possible misuses of exception handling, which may arise from the fact that exception declarations have scope, and that each evaluation of a generative exception binding creates a distinct exception. Consider a simple example:

```
exception  E of bool
fun f(x) =
     let exception E of int
      in if x > 100 then raise E(x) else x+1
     end
val z = f(200) handle E(true) => 500 | E(false) => 1000
```

The program is well-typed, but useless. The exception bound to the outer `E` is distinct from that bound to the inner `E`; thus the exception raised by `f(200)`, with excepted value 200, could only be handled by a handler within the scope of the inner exception declaration—it will not be handled by the handler in the program, which expects a boolean value. So this exception will be reported at top level. This would apply even if the outer exception declaration were also of type int; the two exceptions bound to `E` would be distinct.

On the other hand, if the last line of the program is changed to

```
f(200) handle _ => 500
```

then the exception will be caught, and the value 500 returned. A universal handling rule (i.e. _ or a variable-identifier) catches any exception—even one exported from the scope of the declaration of the associated exception name—but cannot examine the excepted value carried by the exception constructor, since the type of this value cannot be statically determined.

Even a single textual exception binding—if for example it is declared within a recursively defined function—may bind distinct exceptions to the same identifier. Consider another useless program:

```
fun f(x) =
     let exception E
      in if p(x) then a(x)
                 else if q(x) then f(b(x)) handle E => c(x)
                              else raise E
     end
val z = f v
```

Now if p(v) is false but q(v) is true, the recursive call will evaluate f(b(v)). Then if both p(b(v) and q(b(v)) are false, this evaluation will raise `E`. But this exception will not be handled, since the exception raised is that which is bound to `E` by the inner—not outer—evaluation of the exception declaration.

These pathological examples should not leave the impression that exceptions are hard to use or to understand. The rough and ready rule of Section 9 will almost always give the correct understanding.

Chapter 10

Reference values

References are cells whose contents may be changed after creation by assignment. The `ref` "datatype" constructor, and its corresponding value constructor, are almost as if defined by the declaration

```
datatype 'a ref = ref of 'a
```

Thus, a reference whose initial contents are the string `"abc"` may be created by `val r = ref "abc"`. Subsequently, the contents of `r` may be altered by assignment: `r := "def"`. The contents of a reference may be examined by using the `ref` constructor in a pattern:

```
let val (ref s) = r
 in print s
end
```

The function `!` is defined to take the contents of a reference; that is,

```
fun ! (ref x) = x
```

References are not fully polymorphic; see Chapter 11.

Formally, we say that phrases in ML are evaluated in the presence of an *environment* E and a *store* S. The effect on E of evaluating declarations, expressions, etc. is described in Chapter 3. Here we summarize the effect on S.

The store S maps reference values to their contents. Evaluation of an expression in the store S yields, depending on the form of the expression,

ref exp	exp is evaluated in S, producing a value v and a store S'; the reference value r is returned with the store $S' + \{r \mapsto v\}$.

$\mathrm{exp}_1\ \mathrm{exp}_2$ — exp_1 is evaluated in S yielding the function f_1 and store S'; exp_2 is evaluated in S' yielding v_2 and S''; finally the body of f_1 is evaluated with its variable bound to v_2, in the store S'', yielding the result v and the store S'''.

`op :=` $(\mathrm{exp}_1, \mathrm{exp}_2)$ — The expression $(\mathrm{exp}_1, \mathrm{exp}_2)$ is evaluated in S, yielding the pair (r, v) and the store S'; then the unit value $\{\}$ is returned with the store $S' + \{r \mapsto v\}$.

$\{$ lab_1 `=` exp_1 `,` ___ `,` lab_n `=` exp_n $\}$ — exp_1 is evaluated in S, yielding v_1 and the store S_1; then each exp_i is evaluated in S_{i-1}, yielding v_i and the store S_i; then the record $\{\mathrm{lab}_1 = v_1, ..., \mathrm{lab}_n = v_n\}$ is returned with the store S_n. Note that the expressions are evaluated in the sequence they are written, not in alphabetical order of the labels.

`raise` exp — exp is evaluated in S, returning v and S'; then the exception-packet (v, S') is raised.

exp `handle` match — exp is evaluated; if exp returns a value v with state S', then v is returned with S'. If exp raises an exception-packet (e, S'') then the match is applied to e in the state S''. If the match fails, then (e, S'') is raised (as the value of the `handle` expression). If the match succeeds, then the resulting value is returned.

Matching a pattern to a value has no effect on the store. Evaluating a value binding has an effect on the store just from the evaluation of the constituent expressions. Evaluation of type, datatype, or exception bindings has no effect on the store.

Chapter 11

Reference and exception types

The treatment of references and exceptions with "open" types is based on the fact that the contents of a reference cell cannot be constrained to be polymorphic, and so must be considered to be monomorphic. The following example illustrates the problem.

```
let val s = ref (fn x => x)
 in s := (fn x => x+1); (!s) true
end
```

If s were given the polymorphic type $\forall\alpha.(\alpha \to \alpha)\mathbf{ref}$, then this expression would type-check, permitting an obvious type error. To prevent this, we insist that the type of an applied occurrence of the ref constructor should always be given a "ground" type (one with no locally-bound type variables).

However, functions whose application can create reference variables can still have polymorphic types of a restricted kind. Consider the declaration

```
val F = fn x => let val r = ref x
                 in !r
                end
```

Here the function F can be given polymorphic type $\forall\alpha^1.(\alpha^1 \to \alpha^1)$ where α^1 is a special kind of type variable called a *weak* type variable (the superscript "1" indicates that there is one lambda abstraction suspending the creation of the ref cell). When F is applied to an argument, a reference value of type α^1 is created, and hence this weak type variable must be instantiated to a ground type. This means that an expression like `(F nil)` would not be properly typed. In contrast, the type $\alpha^1\mathbf{ref}$ assigned to r is permissible because α^1 is implicitly bound in an outer scope and within the scope of its binding is treated as a constant type.

In ML, weak type variables will be written `'1a`, `'2a`, etc., where the integer after the apostrophe denotes the level of suspension.

Exception declarations raise similar problems, which are handled by an analogous use of weak type variables.

Chapter 12

Modules

ML provides a powerful module system, which can be used to partition programs along clean interfaces.

12.1 Structures

In its simplest form, a module is (syntactically) just a collection of declarations viewed as a unit, or (semantically) the environment defined by those definitions. This is one form of a *structure–expression*: `struct` dec `end`. For example, the following structure–expression represents an implementation of stacks:

```
struct
   datatype 'a stack = Empty | Push of 'a * 'a stack
   exception Pop and Top
   fun empty(Empty) = true | empty _ = false
   val push = Push
   fun pop(Push(v,s)) = s | pop(Empty) = raise Pop
   fun top(Push(v,s)) = v | top(Empty) = raise Top
end
```

Structure–expressions and ordinary expressions are distinct classes; structure–expressions may be bound using the `structure` keyword to structure–names, while ordinary expressions are bound using `val` to value–variables. The form of a `structure` binding is as follows:

`structure` name = structure–expression

Thus, we might make a structure Stack using the structure–expression shown above:

```
structure Stack = struct
```

```
    datatype . . .
    exception . . .
     . . .
  end
```

The environment E that binds the identifiers `stack`, `Pop`, `empty`, etc. is now itself bound to the structure–identifier `Stack`. To refer to names in E, *qualified identifiers* must be used. A qualified identifier consists of a structure–name, a dot, and the name of a structure component, e.g. `Stack.empty` (a value), `Stack.stack` (a type), `Stack.Pop` (an exception), etc.

Structure closure: In order to isolate the interface between a structure and its context, a `struct` phrase is not allowed to contain global references to types, values, or exceptions, except for pervasive primitives of the language like `int`, `nil`, etc. It can, however, contain global references to other structures, signatures, and functors, including qualified names referring to compenents (values, types, etc.) of other structures.

There are three forms of structure-expression:

1. An environment enclosed in `struct` ... `end` (as above),
2. An identifer that has been previously bound in a `structure` declaration, and
3. A functor application `F(`*str*`)`, where `F` is the name of a functor and *str* is a structure expression.

Thus, the declaration `structure~Pushdown~=~Stack` binds the name `Pushdown` to the same structure that `Stack` is bound to; here, `Stack` is an example of the second kind of structure expression.

12.1.1 Accessing structure components

The bindings making up a structure define named *components* of the structure, as in a record. To refer to such components we use qualified names, which are formed by appending a period followed by a component name to the name of a structure. For instance, `Stack.empty` refers to the function `empty` defined in the structure `Stack`. If the qualified name designates a substructure of a structure, then it too has components; *e.g.* `A.B.x` denotes the component `x` of the substructure `B` of a structure `A`.

Qualifiers can be attached only to names; they do not apply to other forms of structure expressions. Qualified names are treated as single lexical units; the dot is not an infix operator.

Direct access to the bindings of a structure is provided by the `open` declaration, which is analagous to the "with" clause of Pascal. For example, in the scope (determined in the usual way) of the declaration

```
open Stack
```

the names `stack`, `empty`, `pop`, etc. refer to the corresponding components of the `Stack` structure. It is as though the body of the structure definition had been inserted in the program at that point, except that the bindings are not recreated, but are instead simply "borrowed" from the opened structure. `open` declarations follow the usual rules for visibility, so that if `A` and `B` are two structures containing a binding for `x` (of the same flavor, of course), then after opening both `A` and `B` with the declaration

```
open A
open B
```

the unqualified identifier `x` will be equivalent to `B.x`. The `x` component of `A` can still be referred to as `A.x`, unless `B` also contains a substructure named `A`.

Qualified identifiers do not have infix status. If `+` is declared infix in a structure `A`, the qualified identifier `A.+` is not an infix identifier. However, when an identifier is made visible by opening a structure, it retains its infix status, if any. AMBIGUOUS.

The declaration `open A B C` is equivalent to `open A; open B; open C`.

12.1.2 Evaluating structure expressions

The evaluation of a structure expression *str* depends on its form, and assumes a current structure environment SE that binds structures and functors to names. Informally, evaluation proceeds as follows:

1. If *str* is an encapsulated declaration (*i.e.* `struct`...`end`), then the body declarations are evaluated relative to SE and the *pervasive* value, exception, and type environments of ML (that is, the environments binding the built-in primitives of the language). The resulting environment is packaged as a structure and returned. The evaluation of value bindings may have an effect on the store (the mapping of references to contents); the new store is returned as well, to be used in subsequent expression evaluations.

2. If *str* is a simple name, then its binding in SE is returned. If it is qualified name, then it is used as an access path starting with SE and the designated substructure is returned.

3. If *str* is a functor application `F(` *str′* `)`, where the functor `F` is declared by `functor F ( A ) =` *body*, the parameter structure *str′* is evaluated in SE yielding structure s_1; then the "body" of the definition of `M`, which is a structure expression, is evaluated in $SE + \{A \mapsto s_1\}$. In other words, functor applications are evaluated in a conventional call-by-value fashion.

12.1.3 Evaluating structure declarations

To evaluate a simple structure declaration, one evaluates the defining structure expression in the current environment SE and returns the binding of the name of the left hand side to the resulting structure. If evaluation of a structure expression raises an (untrapped) exception, then the declaration has no effect.

12.1.4 Structure equivalence

For certain purposes, such as checking sharing constraints (Section ??) we must be able to determine whether two (references to) structures are equal or "the same." Here structures are treated somewhat like datatypes; each evaluation of an encapsulated declaration or functor application creates a distinct new structure, and all references to this structure are considered equal. Thus after the following declarations:

```
structure S1 = struct ... end
structure S2 = S1
structure S3 = struct val x = 4 end
structure S4 = struct val x = 4 end
```

the names `S1` and `S2` refer to the same structure and are "equal," whereas `S3` and `S4` are different structures and are not equal, even though the right-hand-sides are identical.

12.2 Signatures

It is often useful to explicitly constrain a structure binding to limit the visibility of its fields. This is done with a *signature*, which is to a structure binding as a type constraint is to a value binding. For example, we might write a signature for the `Stack` module as

```
sig type 'a stack
    exception Pop and Top
    val Empty : 'a stack
    val push : 'a * 'a stack -> 'a stack
    val empty : 'a stack -> bool
    val pop : 'a stack -> 'a stack
    val top : 'a stack -> 'a
end
```

The signature mentions the structure components that will be visible outside the structure.

Signatures may be bound to identifiers by a signature declaration,

`signature` *sig-Id* = *sig-expr*

where *sig-Id* is an identifier and *sig-expr* is a signature expression—either a `sig`...`end` phrase or a previously bound signature identifier. Thus, the signature above could be bound to the identifier `STACK` by the declaration

```
signature STACK =
   sig type 'a stack
       exception Pop and Top
. . .
   end
```

A signature can be used to constrain a structure by including it in a structure declaration:

`structure` *str-id* : *sig-expr* = *str*

For example, we could write

```
structure Stack1 : STACK = Stack
```

Now the constructor `Push` is not a visible component of the `Stack1` structure, since it doesn't appear in the signature; the qualified identifier `Stack1.Push` is erroneous. Furthermore, since `stack` is mentioned in the signature only as a `type` constructor and not as a `datatype` constructor, the identifier `Stack1.stack` is usable as a type but not a datatype. Finally, since the constructor `Empty` is mentioned as a `val` in the signature, but not as a constructor (*i.e.* as part of a datatype specification), then `Stack1.Empty` may be applied as a function but not matched in a pattern.

There are many signatures that can match the structure `Stack`. One of the "broadest" is

```
structure Stack2 : sig
       datatype 'a stack = Empty | Push of 'a * 'a stack
       exception Pop and Top
       val empty : 'a stack -> bool
       val push : 'a * 'a stack -> 'a stack
       val pop : 'a stack -> 'a stack
       val top : 'a stack -> 'a
   end
= Stack
```

and the "narrowest" is

```
structure Stack3 : sig end = Stack
```

Now, the structure `Stack2` is equivalent to `Stack`; it is the "same" structure, and all the same fields are visible. The structure `Stack3` has no components; there are no qualified identifiers beginning with `Stack3.` However, `Stack3` is the "same" for structure-equivalence purposes as `Stack`, `Stack1`, and `Stack2`; signature constraints do not change the identity of a structure, just which fields are visible.

Appendix A

Syntax of the full language

ide	→	ID	*symbolic or alphabetic*
		`*`	`*` *is legal as a value-identifer*
		=	`=` *may be used but not rebound*
opid	→	ide	
		`op` ide	*removes infix status*
qualid	→	ide	
		ID . qualid	
ident	→	opid	
		qualid	
lab	→	ID	
		INT	*numeric labels*
const	→	INT	
		REAL	
		STRING	
		()	
		ident	*nullary constructor*
exp	→	ident	*variable*
		const	
		# lab	*field selector*
		{ lab = exp , $\underline{\ 0\ }$, lab = exp }	*record*
		(exp , $\underline{\ 2\ }$, exp)	*tuple*
		(exp ; $\underline{\ 1\ }$; exp)	*sequence*
		`[` exp , $\underline{\ 0\ }$, exp `]`	*list*
		`let` decs **`in`** expseq **`end`**	*local declaration*
		exp exp	*application; left–associative*
		exp ide exp	*infixed application*
		exp : ty	*type constraint*
		exp **`andalso`** exp	*conjunction*

		exp `orelse` exp	*disjunction*
		`fn` match	*function*
		`case` exp `of` match	*case expression*
		`while` exp `do` exp	*iteration*
		`if` exp `then` exp `else` exp	*conditional*
		exp `handle` match	*handle exception; right–associative*
		`raise` exp	*raise exception*
match	→	pat `=>` exp | $\underline{\ 1\ }$ | pat `=>` exp	
apat	→	ident	*variable binding*
		const	*constant pattern*
		`_`	*wildcard*
		(pat)	
		(pat , $\underline{\ 2\ }$, pat)	*tuple*
		{ patfield , $\underline{\ 0\ }$, patfield }	*record*
		{ patfield , $\underline{\ 0\ }$, patfield , ... }	*flexible record*
		`[` pat , $\underline{\ 0\ }$, pat `]`	*list*
pat	→	apat	*atomic*
		ident apat	*construction; left–associative*
		pat ide pat	*infixed construction*
		pat : ty	*type constraint*
		opid constraint `as` pat	*layered*
patfield	→	lab = pat	*normal*
		ID	*abbreviation*
		ID `as` pat	*abbreviation*
ty	→	`tyvar`	*type variable*
		{ lab : ty , $\underline{\ 0\ }$, lab : ty }	*record type*
		(ty)	
		(ty , $\underline{\ 2\ }$, ty) qualid	*type construction*
		ty qualid	*unary type construction*
		qualid	*nullary type construction*
		ty `*` $\underline{\ 2\ }$ `*` ty	*typle type*
		ty `->` ty	*function type; right–associative*
vb	→	pat = exp	*simple*
		vb `and` $\underline{\ 1\ }$ `and` vb	*multiple*
constraint	→		*absent*
		: ty	*type constraint*
rvb	→	opid constraint = `fn` match	*simple recursive*
		rvb `and` $\underline{\ 1\ }$ `and` rvb	*mutually recursive*
clause	→	opid apat $\underline{\ 1\ }$ apat constraint = exp	*prefix*
		pat ide pat constraint = exp	*infix*
fb	→	clause | $\underline{\ 1\ }$ | clause	*clausal function*
		fb `and` $\underline{\ 1\ }$ `and` fb	*mutually recursive*
tb	→	tyvars ID = ty	*simple*
		tb `and` $\underline{\ 1\ }$ `and` tb	*multiple*

tyvars	→		*absent*
		`tyvar`	*single*
		(`tyvar` , $\underline{\ 2\ }$, `tyvar`)	*multiple*
db	→	tyvars ID = constr $\mid$ $\underline{\ 1\ }$ $\mid$ constr	*simple*
		db `and` $\underline{\ 1\ }$ `and` db	*mutually recursive*
constr	→	opid	*nullary (constant)*
		opid `of` ty	*unary*
eb	→	ide	*nullary (constant)*
		ide `of` ty	*unary*
		ide = qualid	*re–naming*
		eb `and` $\underline{\ 1\ }$ `and` eb	*multiple*
ldec	→	`val` vb	*value declaration*
		`val` `rec` rvb	*recursive value declaration*
		`fun` fb	*function declaration*
		`type` tb	*type declaration*
		`datatype` db	*datatype declaration*
		`exception` eb	*exception declaration*
		`local` ldec `in` ldec `end`	*local declaration*
		`open` qualid $\underline{\ 1\ }$ qualid	*structure visibility*
		fixity ide $\underline{\ 1\ }$ ide	*directive*
		ldec ldec	*declaration sequence*
		ldec `;`	*optional semicolon*
fixity	→	`infix` INT	*declare infix, left–associative*
		`infix`	*declare infix, precedence 0*
		`infixr` INT	*infix, right associative*
		`infixr`	*infix, right assoc., prec. 0*
		`nonfix`	*cancel infix status*
sgn	→	ID	
		`sig` specs `end`	
specs	→	spec	
		specs spec	
		specs ;	
spec	→	`structure` ID : sgn `and` $\underline{\ 1\ }$ `and` ID : sgn	
		`datatype` db `and` $\underline{\ 1\ }$ `and` db	
		`type` tyvars ID `and` $\underline{\ 1\ }$ `and` tyvars ID	
		`val` ide : ty `and` $\underline{\ 1\ }$ `and` ide : ty	
		`exception` exnspec `and` $\underline{\ 1\ }$ `and` exnspec	
		`sharing` sharspec `and` $\underline{\ 1\ }$ `and` sharspec	
exnspec	→	ide	
		ide `of` ty	
sharspec	→	qualid = $\underline{\ 2\ }$ = qualid	

This description is at present incomplete, as it is missing the grammar rules for structures and functors.

Appendix B

The standard library

A standard set of values, types, exceptions, etc. are *pervasive*—they are in the initial environment and available in every structure. This *standard library* is grouped into structures; each structure deals with the operations on one or two abstract types. The signature of each of these modules, with an informal explanation of the semantics, is given in this chapter.

```
signature GENERAL =
  sig
    infix 3 o
    infix before
    exception Bind
    exception Match
    exception Interrupt
    exception SystemCall of string
    val callcc : ('a cont -> 'a) -> 'a
    val throw : 'a cont -> 'a -> 'b
    val o : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
    val before : ('a * 'b) -> 'a
    datatype 'a option = NONE | SOME of 'a
    type 'a cont
    type exn
    type unit
    infix 4 = <>
    val = : ''a * ''a -> bool
    val <> : ''a * ''a -> bool
  end

abstraction General : GENERAL
```

The structure `General` contains various miscellaneous and general-purpose values, types, and exceptions. The infix `o` is the function composition operator. The infix `before` evaluates both of its arguments and returns the first one. The exceptions `Bind` and `Match` are automatically raised by patterns that fail to match, as explained in chapter 3. The exception `Interrupt` is raised when the INTERRUPT signal is received (e.g. when the user types Control-C or its equivalent). The functions `callcc` and `throw` and the type `'a cont` are used for explicit manipulation of continuations, as explained nowhere. The standard type `exn` is the type of all exceptions and `unit` is the type of empty records. The `=` and `<>` operators are defined here *pro forma*. And finally, the `option` datatype is one we have often found convenient.

B.1 List

```
signature LIST =
  sig
    infixr 5 :: @
    datatype 'a list = :: of ('a * 'a list) | nil
    exception Hd
    exception Tl
    exception Nth
    exception NthTail
    val hd : 'a list -> 'a
    val tl : 'a list -> 'a list
    val null : 'a list -> bool
    val length : 'a list -> int
    val @ : 'a list * 'a list -> 'a list
    val rev : 'a list -> 'a list
    val map :  ('a -> 'b) -> 'a list -> 'b list
    val fold : (('a * 'b) -> 'b) -> 'a list -> 'b -> 'b
    val revfold : (('a * 'b) -> 'b) -> 'a list -> 'b -> 'b
    val app : ('a -> 'b) -> 'a list -> unit
    val revapp : ('a -> 'b) -> 'a list -> unit
    val nth : 'a list * int -> 'a
    val nthtail : 'a list * int -> 'a list
    val exists : ('a -> bool) -> 'a list -> bool
  end
```

The semantics of this module are defined by the following implementation.

```
abstraction List: LIST =
  struct
    infixr 5 :: @
    infix 6 + -
```

```
    datatype 'a list = :: of ('a * 'a list) | nil
    exception Hd
    fun hd (a::r) = a | hd nil = raise Hd
    exception Tl
    fun tl (a::r) = r | tl nil = raise Tl
    fun null nil = true | null _ = false
    fun length nil = 0 | length (a::r) = 1 + length r
    fun op @ (nil,l) = l | op @ (a::r, l) = a :: (r@l)
    fun rev l = let fun f (nil, h) = h
                      | f (a::r, h) = f(r, a::h)
                in  f(l,nil)
                end
    fun map f = let fun m nil = nil | m (a::r) = f a :: m r
                in  m
                end
    fun fold f = let fun f2 nil = (fn b => b)
                       | f2 (e::r) = (fn b => f(e,(f2 r b)))
                 in  f2
                 end
    fun revfold f l = fold f (rev l)
    fun app f l = (map f l; ())
    fun revapp f l = app f (rev l)
    exception Nth
    fun nth(e::r,0) = e
      | nth(e::r,n) = nth(r,n-1)
      | nth _ = raise Nth
    exception NthTail
    fun nthtail(e,0) = e
      | nth(e::r,n) = nthtail(r,n-1)
      | nth _ = raise NthTail
    fun exists f =
          let fun g nil = false | g (h::t) = f h orelse g t
          in  g
          end
  end
```

B.2 Array

```
signature ARRAY =
  sig
    infix 3 sub
    type 'a array
    exception Subscript
```

```
    val array : int * '1a -> '1a array
    val sub : 'a array * int -> 'a
    val update : 'a array * int * 'a -> unit
    val length : 'a array -> int
    val arrayoflist : 'a list -> 'a array
  end
```

Arrays may be made whose elements are any type. `array(n,x)` returns a new array of n elements, indexed from 0 to $n-1$, initialized to x. `a sub i` returns the i^{th} element of the array a. `update(a,i,z)` sets the i^{th} element of the array a to the value z.

Two arrays are equal if and only if they are the same array (created with the same call to `array`); except that all arrays of length 0 may be equal to each other, depending on the implementation.

The following implementation defines the semantics of arrays, though in practice arrays are implemented much more efficiently.

```
abstraction Array : ARRAY =
  struct
   type 'a array = 'a ref list
   exception Subscript
   fun array(0,x) = nil | array(n,x) = ref x :: array(n-1,x)
   fun a sub i = !(nth(a,i)) handle Nth => raise Subscript
   fun update(a,i,z) = nth(a,i) := z handle Nth => raise Subscript
   fun length a = List.length a
   fun arrayoflist l = l
  end
```

B.3 Input/Output

The input/output primitives are intended as a simple basis that may be compatibly superseded by a more comprehensive I/O system that provides for streams of arbitrary type or a richer repertoire of I/O operations. The IO structure contains all I/O primitives; this structure will in all implementation match (with thinning) the BASICIO signature provided below, but may contain other primitives as well.

```
signature BASICIO =
  sig
    type instream
    type outstream
    exception Io of string
    val std_in : instream
    val std_out : outstream
```

```
    val open_in : string -> instream
    val open_out : string -> outstream
    val close_in : instream -> unit
    val close_out : outstream -> unit
    val output : outstream -> string -> unit
    val input : instream -> int -> string
    val lookahead : instream -> string
    val end_of_stream : instream -> bool
  end
```

The type `instream` is the type of input streams and the type `outstream` is the type of output streams. The exception `Io` is used to represent all of the errors that may arise in the course of performing I/O. The value associated with this exception is a string representing the type of failure. In general, any I/O operation may fail if, for any reason, the host system is unable to perform the requested task. The value associated with the exception should describe the type of failure, insofar as this is possible.

The standard prelude binds `std_in` to an instream and `std_out` to an outstream. For interactive ML processes, these are expected to be associated with the user's terminal. However, an implementation that supports the connection of processes to streams may associate one process's `std_in` to another's `std_out`.

The `open_in` and `open_out` primitives are used to associate a disk file with a stream. The expression `open_in(s)` creates a new instream whose producer is the file named `s` and returns that stream as a value. Similarly, `open_out(s)` creates a new `outstream` associated with the file `s`, and returns that stream.

The `input` primitive is used to read characters from a stream. Evaluation of `input s n` causes the removal of `n` characters from the input stream `s`. If fewer than `n` characters are currently available, then the ML system will block until they become available from the producer associated with `s`[1]. If the end of stream is reached while processing an `input`, fewer than `n` characters may be returned. Attempting `input` from a closed stream raises `Io`.

The function `lookahead(s)` returns the next character on `instream s` without removing it from the stream. Input streams are terminated by the `close_in` operation. This primitive is provided primarily for symmetry and to support the re-use of unused streams on resource-limited systems. The end of an input stream is detected by `end_of_stream`, a derived form that is defined as follows:

```
fun end_of_stream(s) = (lookahead(s)="")
```

[1]The exact definition of "available" is implementation-dependent. For instance, operating systems typically buffer terminal input on a line-by-line basis so that no characters are available until an entire line has been typed.

Characters are written to an `outstream` with the `output` primitive. The string argument consists of the characters to be written to the given outstream. The function `close_out` is used to terminate an output stream. Any further attempts to output to a closed stream cause `Io` to be raised.

```
signature IO =
  sig
    type instream
    type outstream
    exception Io of string
    val std_in : instream
    val std_out : outstream
    val open_in : string -> instream
    val open_out : string -> outstream
    val open_append : string -> outstream
    val open_string : string -> instream
    val close_in : instream -> unit
    val close_out : outstream -> unit
    val output : outstream -> string -> unit
    val input : instream -> int -> string
    val input_line : instream -> string
    val lookahead : instream -> string
    val end_of_stream : instream -> bool
    val can_input : instream -> int
    val flush_out : outstream -> unit
    val is_term_in : instream -> bool
    val is_term_out : outstream -> bool
    val set_term_in : instream * bool -> unit
    val set_term_out : outstream * bool -> unit
    val execute : string -> instream * outstream
    val exportML : string -> bool
    val exportFn : string * (string list * string list -> unit) -> unit
    val use : string -> unit
    val use_stream : instream -> unit
    val reduce : ('a -> 'b) -> ('a -> 'b)
    val mtime : instream -> int
(* the following are temporary components *)
    val reduce_r : ((unit -> unit) -> (unit -> unit)) ref
    val cleanup : unit -> unit
    val use_f : (string -> unit) ref
    val use_s : (instream -> unit) ref
  end

structure IO : IO
```

In addition to the basic I/O primitives, provision is made for some extensions that are likely to be provided by many implementations. The functions listed above are provided by Standard ML of New Jersey.

The function `execute` is used to create a pair of streams, one an `instream` and one an `outstream`, and associate them with a process. The string argument to `execute` is the operating-system command that starts the process.

The function `flush_out` ensures that the consumer associated with an `out_stream` has received all of the characters associated with that stream. It is provided primarily to allow the ML user to circumvent undesirable buffering characteristics that may arise in connection with terminals and other processes. All output streams are flushed when they are closed, and in many implementations an output stream is flushed whenever a newline character is written if that stream is connected to a terminal.

The function `can_input` returns the number of characters which may be read from its instream argument without blocking. For instance, a command processor may wish to test whether or not a user has typed ahead in order to avoid an unnecessary prompt. The exact definition of "currently available" is implementation specific, perhaps depending on such things as the processing mode of a terminal.

The `input_line` primitive returns a string consisting of the characters from an `instream` up through, and including, the next end of line character. If the end of stream is reached without reaching an end of line character, all remaining characters from the stream (*without* an end of line character) are returned.

Files may be open for output while preserving their contents by using the `open_append` primitive. Subsequent `output` to the outstream returned by this primitive is appended to the contents of the specified file.

Basic support for the complexities of terminal I/O are provided. The pair of functions `is_term_in` and `is_term_out` test whether or not a stream is associated with a terminal; and `set_term_in` and `set_term_out` tell the ML system that a stream is (or is not) a terminal. These functions are especially useful with `std_in` and `std_out` because they are opened as part of the standard prelude. A terminal may be associated with a stream using the ordinary `open_in` and `open_out` functions; the naming convention to do this is implementation-dependent.

Given a name of a file, `use` compiles and executes its contents as if they were typed into the top-level prompt of the interactive system. `use` may be nested recursively. Similarly, `use_stream` compiles an already-opened instream.

`exportML` creates an executable file whose name is given by the argument. When this file is executed, it is an ML system in exactly the same state as the one that wrote the file. For example, the command `(exportML "foo"; print "Hello");` writes a file that, when executed, prints `Hello` and then returns to the top-level prompt. exportML returns true when the executable file is run, and false when simply returning.

`exportFn` creates an executable file whose name is given by the first argument. When this file is executed, it is an ML system that calls the function given as the second argument, then exits. The ML system created will not have a compiler or a top-level, so it will be significantly more compact. The command-line arguments and environment are passed as the string list arguments to the function that is called. `exportFn` terminates execution of the ML system that called it.

B.4 Bool

```
signature BOOL =
  sig
    datatype bool = true | false
    val not: bool -> bool
    val print: bool -> bool
    val makestring: bool -> string
  end
```

These are quite straightforward, and can be defined as follows:

```
abstraction Bool: BOOL =
  struct
    datatype bool = true | false
    fun not true = false | not false = true
    fun makestring true = "true" | makestring false = "false"
    fun print b = (output(std_out, makestring b); b)
  end
```

B.5 ByteArray

```
signature BYTEARRAY =
  sig
    infix 3 sub
    eqtype bytearray
    exception Subscript
    exception Range
    val array : int * int -> bytearray
    val sub : bytearray * int -> int
    val update : bytearray * int * int -> unit
    val length : bytearray -> int
    val extract : bytearray * int * int -> string
    val fold : ((int * 'b) -> 'b) -> bytearray -> 'b -> 'b
    val revfold : ((int * 'b) -> 'b) -> bytearray -> 'b -> 'b
```

```
    val app : (int -> 'a) -> bytearray -> unit
    val revapp : (int -> 'b) -> bytearray -> unit
  end
```

Byte arrays are just like arrays of integers, with the restriction that the values of the component integers must be between 0 and 255. The intent is that the implementation may store them more efficiently than the equivalent array.

Note that, by default, the ByteArray structure is present but not opened in the initial environment. The declaration `open ByteArray` may be used to open it. The use of ByteArray is discouraged; future versions of the compiler may not support it, or (for example) debuggers might not support it.

The semantics can be defined by this implementation:

```
abstraction ByteArray : BYTEARRAY =
  struct
    infix 3 sub
    type bytearray = int array
    exception Subscript = Array.Subscript
    exception Range
    fun check x = if x<0 orelse x>255 then raise Range else ()
    fun array(i,x) = (check x; Array.array(i,x))
    val length = Array.length
    fun update(a,i,x) = (check x; Array.update(a,i,x))
    val op sub = Array.sub
    fun extract(b,i,0) = if i<0 orelse i>length(b)
                          then raise Subscript  else ""
      | extract(b,i,n) = chr(b sub i) ^ extract(b,i,n-1)
    val fold =  . . .
    val revfold =  . . .
    val app = ...
    val revapp = ...
  end
```

B.6 Integer

```
signature INTEGER =
  sig
    infix 7 * div mod
    infix 6 + -
    infix 4 > < >= <=
    exception Div
    exception Overflow
    type int
    val ~ : int -> int
```

```
    val * : int * int -> int
    val div : int * int -> int
    val mod : int * int -> int
    val + : int * int -> int
    val - : int * int -> int
    val >  : int * int -> bool
    val >= : int * int -> bool
    val <  : int * int -> bool
    val <= : int * int -> bool
    val min : int * int -> int
    val max : int * int -> int
    val abs : int -> int
    val print : int -> unit
    val makestring : int -> string
  end
```

This should be mostly self-explanatory. The function `div` raises `Div` on divide by zero, otherwise `Overflow` if the result doesn't fit; similarly `mod` may raise `Div` or `Overflow`. Other operators may raise `Overflow` if the result doesn't fit into their representation. Standard ML of New Jersey uses finite precision signed 31-bit integers, which can represent a range from -2^{30} to $2^{30}-1$.

B.7 Real

```
signature REAL =
  sig
    infix 7 * /
    infix 6 + -
    infix 4 > < >= <=
    type real
    exception Floor and Sqrt and Exp and Ln
    exception Real of string
    val ~ : real -> real
    val + : (real * real) -> real
    val - : (real * real) -> real
    val * : (real * real) -> real
    val / : (real * real) -> real
    val > : (real * real) -> bool
    val < : (real * real) -> bool
    val >= : (real * real) -> bool
    val <= : (real * real) -> bool
    val abs : real ->  real
    val real : int -> real
```

```
    val floor : real -> int
    val truncate : real -> int
    val ceiling : real -> int
    val sqrt : real -> real
    val sin : real -> real
    val cos : real -> real
    val arctan : real -> real
    val exp : real -> real
    val ln : real -> real
    val print : real -> unit
    val makestring : real -> string
  end
structure Real : REAL
```

This should be mostly self-explanatory. Except for the special exceptions `Floor`, `Sqrt`, `Exp`, `Ln`, raised by the functions of the corresponding names, all real-number functions raise only the `Real` exception with some system dependent argument string.

B.8 Ref

```
signature REF =
  sig
    infix 3 :=
    val ! : 'a ref -> 'a
    val := : 'a ref * 'a -> unit
    val inc : int ref -> unit
    val dec : int ref -> unit
  end
```

Reference values are described in chapter 10. The functions `inc` and `dec` can be defined as

```
  fun inc i = i := !i+1
  fun dec i = i := !i-1
```

B.9 String

```
signature STRING =
  sig
    infix 6 ^
    infix 4 > < >= <=
    type string
```

```
    exception Substring
    val length : string -> int
    val size : string -> int
    val substring : string * int * int -> string
    val explode : string -> string list
    val implode : string list -> string
    val <= : string * string -> bool
    val <  : string * string -> bool
    val >= : string * string -> bool
    val >  : string * string -> bool
    val ^  : string * string -> string
    exception Chr
    val chr : int -> string
    exception Ord
    val ord : string -> int
    val ordof : string * int -> int
    val print : string -> string
  end
```

Strings can be explained by the implementation below; of course, in practice a more efficient implementation is used.

```
abstraction String : STRING =
  struct
    infix 6 ^
    infix 4 > < >= <=
    type string = int list
    exception Substring
    val length = List.length
    val size = length
    fun substring(s,0,0) = nil
      | substring(a::b,0,len) = a::substring(b,0,len-1)
      | substring(nil,_,_) = raise Substring
      | substring(a::b,i,len) = substring(b,i-1,len)
    fun explode nil = nil
      | explode (i::l) = [i] :: explode l
    fun implode nil = nil
      | implode (s::l) = s @ implode l
    fun (_::_) > nil = true
      | nil > (_::_) = false
      | (i::r) > (j::s) = Integer.>(i,j) orelse i=j andalso r>s
    fun a <= b = not (a>b)
    fun a < b = b > a
    fun a >= b = b <= a
    val op ^ = op @
```

```
    exception Chr
    fun chr i = if i<0 orelse i>255 then raise Chr else [i]
    exception Ord
    fun ord nil = raise Ord | ord (i::r) = i
    fun ordof(s,i) = nth s handle Nth => raise Ord
    fun print s = (output(std_out,s); s)
  end
```

B.10 Bits

```
signature BITS =
  sig
    type int
    val orb : int * int -> int
    val andb : int * int -> int
    val xorb : int * int -> int
    val lshift : int * int -> int
    val rshift : int * int -> int
    val notb : int * int -> int
  end
structure Bits : BITS
```

The structure Bits allows shifting and masking of integers (viewed as strings of binary digits). This structure is present but *not* opened in the standard environment; its use is discouraged. The right shift (rshift) operator may shift 0's, 1's, or sign bits into the left end of an integer at its discretion.

B.11 System

```
signature SYSTEM =
  sig
    structure Control : CONTROL
    structure Tags : TAGS
    structure Timer : TIMER
    structure Stats : STATS
    structure Unsafe : UNSAFE
    val exn_name : exn -> string
    val version : string
    val interactive : bool ref
    val cleanup : unit -> unit
    val system : string -> unit
    val cd : string -> unit
    val argv : unit -> string list
```

```
    val environ : unit -> string list
  end
structure System : SYSTEM
```

Features of Standard ML of New Jersey that should not be expected in any other implementation of ML are grouped into the System structure, which is present but not opened in the standard environment.

The substructures `Control` of `System` are not documented.

The function `exn_name` returns the name of the exception constructor that was used to build a given exception value. The `version` string indicates which version of Standard ML of New Jersey is running. The variable `interactive` may be set to indicate whether the compiler's input stream should be treated as interactive (i.e. issue primary and secondary prompts, read a line at a time) or non-interactive (i.e. no prompts, read a large block at a time). The `cleanup` function closes all files. The `system` function runs an operating-system (shell) command specified by its argument. `cd` changes the current working directory. `argv` and `environment` return the command-line argument-list and (Unix) environment with which the Standard ML process was created.

Appendix C

Compatibility

The language definition has changed a bit over the past year (1986–87), particularly in the area of exception syntax. Some attempt has been made to allow old programs to continue running.

C.1 Exceptions

Three keywords: `exceptionx`, `raisex`, and `handlex` are provided. They implement the old-style exception mechanism, and are treated as derived forms.

Derived Form	Equivalent Form
`exceptionx` identifier	`exception` Identifier
`exceptionx` identifier : ty	`exception` Identifier `of` ty
`raisex` identifier	`raise` Identifier
`raisex` identifier `with` exp	`raise` Identifier (exp)
`handlex` ident => exp	`handle` Ident => exp
`handlex` ident => exp	`handle` Ident () => exp
`handlex` ident `with` pat => exp	`handle` Ident pat => exp

The derivations for `handlex` are a bit more intricate than shown above. The intent is that any program that works under the old scheme will continue to work if all instances of `exception`, `handle`, and `raise` are changed to `exceptionx`, `handlex`, and `raisex` respectively.

Note that the derived forms change the exception identifier in converting to the standard forms; they capitalize the first letter. This is to simulate the old scheme, in which exception names lived in a different space from value names. This will cause problems in any programs with a capitalized value-name that happens to conflict with a (capitalized or uncapitalized) exception name.