Abstract

This is the HimML reference manual. HimML is a close variant of the Standard ML language, with set-theoretic types, data structures and operators, and an extendible physical unit system.

Keywords: ML, sets, physical units.

Résumé

Ceci est le manuel de référence de HimML. HimML est une proche variante du langage Standard ML, avec des types, des structures de données et des opérateurs ensemblistes, et un système d’unités physiques extensible.

Mots-clés : ML, ensembles, unités physiques.


Bison parser generator Copyright (c) 1986 Free Software Foundation, Inc.

Regular expression matcher Copyright (c) 1986 by University of Toronto
# Contents

1 Overview 3

2 Differences with Standard ML 4
   2.1 Sets ......................................................... 4
       2.1.1 Set and Map Expressions, Comprehensions ................. 5
       2.1.2 Set and Map Patterns ..................................... 10
       2.1.3 Typing Sets and Maps .................................... 11
   2.2 Numbers and Physical Units .................................. 12
   2.3 Other Differences ........................................... 14

3 Core Types 16

4 The Standard Library 23
   4.1 General Purpose Exceptions and Functions .................. 23
   4.2 Lists .......................................................... 25
   4.3 Sets and Maps .................................................. 26
   4.4Refs and Arrays ............................................... 34
   4.5 Strings ........................................................ 35
   4.6 Integer Arithmetic ........................................... 39
   4.7 Numerical Operations ......................................... 41
   4.8 Large Integer Arithmetic .................................... 45
   4.9 Input/Output Arithmetic ...................................... 55
   4.10 Miscellaneous ............................................... 61

5 Running the System 65
   5.1 Starting up .................................................... 65
   5.2 Compiling, Linking, Finding Dependencies ................... 69
   5.3 Debugger ....................................................... 71
   5.4 Profiler ......................................................... 73
   5.5 Separate Compilation and Modules ............................. 75
       5.5.1 Overview .................................................. 75
       5.5.2 Header Files ............................................... 77
       5.5.3 Summary .................................................... 79
   5.6 Editor Support ................................................ 80
   5.7 Bugs ............................................................ 81
   5.8 Common Problems .............................................. 81
       5.8.1 Problems When Installing HimML ....................... 81
       5.8.2 Problems When Running HimML ........................... 82
   5.9 Reporting Bugs, Making Suggestions ........................... 83
Chapter 1

Overview

The HimML language is a close cousin to the Standard ML language [6, 9]. Actually, it is Standard ML without its module system (as of October, 1992), but with some extensions and with a specially designed run-time system. This document is a reference for HimML version 1.0 alpha 20. It does assume some knowledge of the syntax, static and dynamic semantics of the Standard ML language, and builds on this knowledge.

The plan of the document is as follows. In the rest of chapter 1, we quickly describe what makes the originality of HimML. Chapter 2 is devoted to the differences between HimML and Standard ML; they are mainly extensions, but some are mere differences. Chapter 3 lists the types and values provided upon booting the HimML executable; others are defined in source form, with their own documentation. There is no guide to the syntax of HimML, for which the reader is referred to [6], and to the appropriate sections of this report when HimML syntax differs from the Standard ML syntax. Chapter 4 discusses running the system, and also how you can contribute to improve it, either by reporting bugs or suggesting changes.

The run-time system is architectured around the concept of maximal sharing, or systematic hash-consing [4], in which two identical objects in the heap lie exactly at the same address. This makes comparison fast, though memory management may suffer a bit occasionally. However, this approach has a lot of advantages in terms of memory usage (reuse of objects) and of speed (via memoization and computation sharing); these points are addressed in [4].

The main feature added to Standard ML in HimML is the collection of set and map operations, types and syntactic constructs. A set in HimML means a finite collection of objects of the same type, where order and multiplicity are irrelevant. A map looks like a set: it is a finite collection of associations, called maplets, from objects of a type \( t_1 \) to objects of a type \( t_2 \), forming a many-to-one relation. Because maps subsume sets, sets of objects of type \( t \) are just defined in HimML as maps from objects of type \( t \) to the special object \( () \), the empty tuple. The need for sets and maps as basic type constructors in programming languages is advocated in [3]. In HimML, sets and maps form the core of the language, and, thanks to maximal sharing, operations on sets and maps are very fast. It is encouraged to write HimML programs as executable specifications in a set-theoretic framework, for it is easy and in general more efficient than we think it could be in advance.

Another feature added in HimML is its typing of numerical values with a notion of measure units. This scheme allows to detect inconsistencies in scientific programs, where the problem lies not in the structure of data (numbers) but in the nature of what they describe (units).

We shall often refer to the Standard ML language, as defined in [6] and [9]. Sometimes, it will be necessary to distinguish particular features of a prominent implementation of Standard ML, called Standard ML of New Jersey or SML-NJ (unpublished documentation, provided with the 1991 release of SML-NJ).

Note that there is a difference between the sign \( \ldots \), that is used to describe a repetition of objects in the syntax, and the sign \( \ldots \), which is a lexeme, used in tuple and record patterns, and also in extensible types (an extension to Standard ML’s type system).
Chapter 2

Differences with Standard ML

2.1 Sets

The type constructor for maps is noted \(-m>\), and is infixed just like the \(\rightarrow\) function type constructor. When declaring a type \(t1 \to-m> t2\), the type \(t1\) must admit equality (that is, the equality operation \(\_\) must be applicable to elements of type \(t1\)); this is because maps rely on comparison of elements of their domain.

The type of sets of objects of type \(t\) (which admits equality) is \(t\ set\), which is an abbreviation for \(t \to-m> \text{unit}\) (\(\text{unit}\) is the type of the empty tuple \(()\)).

Any map type \(t1 \to-m> t2\) admits equality as soon as \(t2\) admits equality (\(t1\) always admits equality).

Special notations are provided to build sets and maps:

\[\{x => y, x' => y', x'' => y''\}\] is an example of a map described by \emph{enumeration}. This maps \(x\) to \(y\), \(x'\) to \(y'\), and \(x''\) to \(y''\). If some of \(x, x'\) or \(x''\) are the same, for instance \(x=x'\), then the resulting map may associate \(x\) with either \(y\) or \(y'\) nondeterministically.

As a shortcut, we can write \(x\) instead of \(x => ()\), so that \(\{x, y, z\}\) is the set containing exactly \(x\), \(y\) and \(z\).

\[\{f(x) => g(x) \mid x \text{ in set } s\}\] is an example of a map described by \emph{comprehension}. The \(x \text{ in set } s\) is the \emph{domain description}. In this example, all maplets \(f(x) => g(x)\) are formed for \(x\) ranging over the set \(s\) (actually \(s\) may be a map, but only its domain matters), and are collected to form a map. The \(x \text{ in set } s\) domain description is a particular case of the \emph{in map domain description}: the domain description \(x => y \text{ in map } m\) makes \(x\) range over the domain of \(m\), and simultaneously binds \(y\) to the image of \(x\) by \(m\), so \(x \text{ in set } s\) is syntactic sugar for \(x => _\text{ in map } s\). In \(x => y \text{ in map } m\), the \(\Rightarrow y\) part may be omitted when \(y\) is the constant \((\)\).

Other domain descriptions are of the form \(x \text{ in list } l\), which makes \(x\) sweep through the list \(l\), and \(x \text{ sub map } m\), which makes \(x\) sweep through the set of all maps that are included in the map \(m\). The order in which \(s\) are swept through (with \(\text{in set in map or sub map}\)) is not specified, though sweeping through the same set twice in the same HimML session will be done in the same order. This is important since if there are two values of \(x\) that make \(f(x)\) equal, then only the last is retained in the result. To reverse this behavior (taking the first of the two values), we use the notation \(<\{\), as in \(<\{f(x) => g(x) \mid x \text{ in set } s\}\). Domain descriptions may be combined with the \emph{and} connector. For example, the expression

\[\{f(x, y) => g(x) \mid x \text{ in set } s1 \text{ and } y \text{ sub map } S\}\]

maps \(f(x, y)\) to \(g(x)\) for all \(x\) in the domain of the map \(s1\), and for all submaps \(y\) of \(S\). This is used to build cross products, like in \(\{(x, y) => () \mid x \text{ in set } s1 \text{ and } y \text{ in set } s2\}\). The order in which \(s1\) and \(s2\) are swept through is nondeterministic, except that the same domain description will always sweep through the same data in the same order during one HimML session, and that if we look at all the values of \(y\) produced for one given value of \(x\), they are all produced in the same order as if there were only the descriptor \(y \text{ in set } s2\) (same thing if we fix \(y\) and look at the sequence of values for \(x\)).

Domain descriptions may be filtered with a \emph{such that} phrase. For example, the expression

\[\{(x, y) => x+y \mid x \text{ in set } s1 \text{ and } y \text{ in set } s2 \text{ such that } x*y=3\}\]
maps only pairs \((x, y)\) such that \(x + y = 3\) to their sum.

As for definition by enumeration, writing \(f(x)\) instead of \(f(x) \rightarrow ()\) is allowed. For instance,

\[
\{x \mid x \text{ sub map } f\}
\]
is the set of all submaps of the map \(f\) (the powerset of \(f\) when \(f\) is a set).

## 2.1.1 Set and Map Expressions, Comprehensions

In general, we have the following form of set and map comprehension:

\[
\{\text{expression} \mid \text{expression}' \mid \mid \mid \mid \text{pattern}_1 \mid \text{pattern}'_1\} (\text{ in map } \mid \text{ in set } \mid \text{ in list } \mid \text{ sub map } \mid \text{expression}_1 \mid \text{expression}'_1 \mid \\text{pattern}_1 \mid \text{pattern}'_1\}
\]

where signs and words in typewriter style are meant literally, where parentheses serve as grouping marks, brackets enclose an optional construct, a vertical bar means an alternative and a star denotes zero or more occurrences of a construct. The \([ \mid \text{ pattern}'_i\] forms are only allowed if followed by \text{in map}. The expressions \text{expression}, \text{expression}'\), \text{expression}" and \text{expression}" are ordinary HimML expressions. \text{pattern}_i\ and \text{pattern}'_i\ are HimML patterns, that are essentially Standard ML patterns, augmented with special set and map patterns (see section 2.1.2).

This expression executes as follows:

- First the expressions \text{expression}_i\ are evaluated, giving domains we note \(D_i\). Let \(IN_i\ be the keyword \text{in map, in set, in list or sub map that is between pattern}_i\ and \text{expression}_i\.

\(\text{If } IN_i \text{ is in list, the typing system ensures that } D_i \text{ is a list, then let } L_i = D_i. \text{If } IN_i \text{ is in set or in map, } D_i \text{ is a map (this is enforced by the typing system), and we let } L_i \text{ be the list of elements in the domain of } D_i, \text{ in an order specified in the next paragraph. If } IN_i \text{ is sub map, } D_i \text{ is a map again, and we let } L_i \text{ the list of submaps of } D_i \text{ (i.e., maps having a domain included in the domain of } D_i, \text{ and mapping all } x \text{ to the same } y \text{ than } D_i), \text{ in an order specified in the next paragraph.}

\text{Call } n_i \text{ the length of the list } L_i.

- then the lists \(L_i\) are swept through, in a cross-product fashion if we used the separator \mid between the maplet and domain description parts of the comprehensions, or in a parallel fashion (first elements first, then all the second elements, etc. no notion of concurrency here) if the separator was \mid \mid \mid \mid \mid. \text{This yields elements } l_i, \text{ which are matched again the } \text{pattern}_i, \text{ for all } i, \text{ in a new local environment (in the case of the } \text{in map} \text{ domain descriptor, the pattern } \text{pattern}'_i, \text{ or } () \text{ if absent, are matched against the image of } l_i \text{ by the map } D_i).

- If all matchings succeed, and if the filter \text{expression}" evaluates to \text{true} in the new environment (if the such that part is not present, we take \text{expression}" to be \text{true}), then \text{expression} and \text{expression}' are evaluated in the new environment (if absent, \text{expression}' is taken to be ()).

- Finally, all resulting maplets are collected into a map. \text{If there is a collision (i.e, expression evaluates to the same value for two runs), then the last value of expression}' is taken. Conceptually, every new maplet overwrites the old ones.

The order in which the domains are swept through has the following properties:

- lists (for in list) are swept through from left to right: \(x \text{ in list } \{a, b, c\} \text{ yields } a, \text{ then } b, \text{ then } c;

- maps (for in set and in map) are swept through in an unspecified order, going through each maplet exactly once. This order depends only on the session (we call session a given HimML process). This means in particular that \(a\) may come before \(b\) in one session, but after \(b\) on the same input data, in another session; however, in the same session, the order will always be the same; this is a weak form of nondeterminism.
Moreover, this order does not depend on the map itself, so that \(a\) and \(b\) are always encountered in the same order in the same session, whether we sweep through \(\{a, b, c\}\) or \(\{x, y, a\}\) for instance. This order is actually a typed instance of the system order \(\leq\), a total order on HimML objects of the same type. The system order has no a priori connection with any other order on numbers or strings, or sets, or whatever.

- the submaps of a map (with \text{sub map}) are swept through in a yet another unspecified order, which may not be the arrangement of submaps in the system order. This order has the same weakly nondeterministic behavior as the \text{in set} and \text{in map} traversals. The set of all submaps is not actually built, to save space.

- The interleaving of traversals is left unspecified in the case of the \(|\) separator. However, the order of individual traversals is left unchanged. Precisely, number from 1 to \(n\), the elements extracted from \(D\), when we have only one domain.

When we have \(m\) domains \(D_1, \ldots, D_m\), we sweep through \(n_1 b_2 \ldots n_m\) elements: index these elements by tuples \((j_1, \ldots, j_m)\), \(j_1 = 1 \ldots n_1, \ldots, j_m = 1 \ldots n_m\). Then, when \(\forall i \neq k \cdot j_i = j'_i\), the element of index \((j_1, \ldots, j_m)\) occurs before the element of index \((j'_1, \ldots, j'_m)\) if and only if \(j_k < j'_k\). We leave unspecified the cases when \(\forall i \neq k \cdot j_i = j'_i\) does not hold.

In the case of the \(||\) separator, all the \(n_i\) are equal, and the element of index \((j_1, \ldots, j_m)\) occurs before the element of index \((j'_1, \ldots, j'_m)\) if and only if \(j_i < j'_i\) for all \(i\).

An alternate form of set or map enumeration uses a \(<\{\) instead of the opening brace:

\[
\begin{align*}
\{ & \text{expression}_1 [ \Rightarrow \text{expression}_1' ]* \\
 & \text{expression}_i [ \Rightarrow \text{expression}_i' ]* 
\end{align*}
\]

The same holds for comprehension:

\[
\{ \text{expression} [ \Rightarrow \text{expression'}] ||| \\
\text{pattern}_1 [ \Rightarrow \text{pattern}_1'](\text{in map}|\text{in set}|\text{in list}|\text{sub map})\text{expression}_1 \\
( \text{and} \text{pattern}_i [ \Rightarrow \text{pattern}_i'](\text{in map}|\text{in set}|\text{in list}|\text{sub map})\text{expression}_i)* \\
[ \text{such that expression'}] 
\]

The semantics are the same, except that overwriting is replaced by “underwriting”: in case of a collision, the first maplet is retained instead of the last.

The comprehension notation has been extended to list comprehensions:

\[
\begin{align*}
\{ & \text{expression} ||| \\
\text{pattern}_1 [ \Rightarrow \text{pattern}_1'](\text{in map}|\text{in set}|\text{in list}|\text{sub map})\text{expression}_1 \\
( \text{and} \text{pattern}_i [ \Rightarrow \text{pattern}_i'](\text{in map}|\text{in set}|\text{in list}|\text{sub map})\text{expression}_i)* \\
[ \text{such that expression'}] 
\end{align*}
\]

collects values of \text{expression} in a list, in the order they are encountered. This subsumes the \text{map} functional of Standard ML:

\[
\begin{align*}
\text{fun map f l} & = [f(x) | x \text{ in list } l] 
\end{align*}
\]

Comprehensions have also been extended to handle logical structures. In this case, they are called \text{quantifications}. We have universal quantification:

\[
\begin{align*}
\text{all} \text{ expression} ||| \\
\text{pattern}_1 [ \Rightarrow \text{pattern}_1'](\text{in map}|\text{in set}|\text{in list}|\text{sub map})\text{expression}_1 \\
( \text{and} \text{pattern}_i [ \Rightarrow \text{pattern}_i'](\text{in map}|\text{in set}|\text{in list}|\text{sub map})\text{expression}_i)* \\
[ \text{such that expression'}] \text{ end} 
\end{align*}
\]
which sweeps through the domains as usual, and returns true if expression always evaluates to true, and false otherwise. Evaluation of the universal quantification stops as soon as expression returns false (all pending elements of the domains are ignored), or when all domains have been traversed. The all quantifier can be seen as a generalization of the andalso logical connective.

The existential quantification is dual (by negation) of the universal one:

\[
\text{exists } expression \mid \mid \mid \\
\text{pattern}_i[ \Rightarrow \text{pattern}_i'][(\text{in map}|\text{in set}|\text{in list}|\text{sub map}) expression_1]
\text{ ( and } \text{pattern}_i[ \Rightarrow \text{pattern}_i'][(\text{in map}|\text{in set}|\text{in list}|\text{sub map}) expression_i])\ast
\text{ [ such that } expression''\ast\text{] end}
\]

This returns true if for some value in the domains expression evaluates to true, and false otherwise. Evaluation of the existential quantification stops as soon as expression returns true (all pending elements of the domains are ignored), or when all domains have been traversed. The exists quantifier can be seen as a generalization of the orelse logical connective.

Close to existential quantification, we have the choice:

\[
\text{some } expression \mid \mid \mid \\
\text{pattern}_i[ \Rightarrow \text{pattern}_i'][(\text{in map}|\text{in set}|\text{in list}|\text{sub map}) expression_1]
\text{ ( and } \text{pattern}_i[ \Rightarrow \text{pattern}_i'][(\text{in map}|\text{in set}|\text{in list}|\text{sub map}) expression_i])\ast
\text{ [ such that } expression''\ast\text{] end}
\]

This looks for the value e of expression in the first environment where all patterni match elements of Di while making expression'' true. All other elements of the domains are ignored. If e is found, then it returns some e. If there is no such expression, NONE is returned. (NONE and SOME are the constructors of the option datatype; see Section 3.3)

So, existential quantification is a special case of choice: exists ... end could be written

\[
\text{case (some } /\text{ldots}/ \text{verb/ end) of } \text{NONE } \Rightarrow \text{false } | \text{SOME } _ \Rightarrow \text{true}
\]

A last form of comprehension generalizes the sequence connector ;. It is iteration:

\[
\text{iterate } expression \mid \mid \mid \\
\text{pattern}_i[ \Rightarrow \text{pattern}_i'][(\text{in map}|\text{in set}|\text{in list}|\text{sub map}) expression_1]
\text{ ( and } \text{pattern}_i[ \Rightarrow \text{pattern}_i'][(\text{in map}|\text{in set}|\text{in list}|\text{sub map}) expression_i])\ast
\text{ [ such that } expression''\ast\text{] end}
\]

It sweeps though the domains in the usual order, and evaluates expression in all matching environments. It is interesting essentially only if expression produces or depends on side-effects. It returns (). This subsumes the app functional of Standard ML:

\[
\text{fun app f l = iterate f(x) | x in list l end}
\]

Corresponding to each form of comprehension, we finally have a form of imperative comprehension, mostly useful with programs having or depending on side-effects:

\[
\{expression[ \Rightarrow expression']
\text{ [ while expression''\ast\text{] end}
\]

This evaluates expression''', and returns the empty set {} if it is false. Otherwise, it evaluates expression'' and if it is true, then expression and expression'. All maplets that we get this way (such that expression'' evaluates to true), are collected in a map, which is returned as soon as expression''' evaluates to false. Collisions are resolved by an overwriting strategy (last maplet wins).

It is syntactic sugar for (expression'' being assumed true when not present, and expression' being assumed () if absent):
let fun collect s=
  if expression"
   then if expression"'
        then collect (s ++ {expression => expression'})
        else collect s
       else s
  in
  collect ()
end

assuming s is a new variable (++ is the infix map overwrite function).
Similarly, the underwriting imperative map comprehension notation is:

{ expression [ => expression' ] |
   while expression" |
   [ such that expression"] }

which is syntactic sugar for:

let fun collect s=
  if expression"
   then if expression"'
        then collect ({expression => expression'} ++ s)
        else collect s
       else s
  in
  collect ()
end

The imperative list comprehension notation is:

[ expression | while expression"
   [ such that expression"] ]

which is syntactic sugar for:

let fun collect l=
  if expression"
   then if expression"'
        then expression :: collect l
        else collect l
       else l
  in
  collect []
end

where l is a new variable, and :: is the infix list constructor (cons is Lisp; note that in HimML as in Standard ML, a::l evaluates a before l).
The imperative universal quantification notation is:

all expression | while expression"
   [ such that expression"]
end

which is syntactic sugar for:
let fun quantify () =
    if expression
    then if expression
        then expression andalso quantify ()
        else quantify ()
    else true
in
    quantify ()
end

The imperative existential quantification notation is:

```
exists expression | while expression
[ such that expression]
end
```

which is syntactic sugar for:

let fun quantify () =
    if expression
    then if expression
        then expression orelse quantify ()
        else quantify ()
    else false
in
    quantify ()
end

The imperative choice notation is:

```
some expression | while expression
[ such that expression]
end
```

which is syntactic sugar for:

let fun quantify () =
    if expression
    then if expression
        then SOME expression
        else NONE
    else quantify ()
in
    quantify ()
end

The imperative iteration notation is:

```
iterate expression | while expression
[ such that expression]
end
```

which is syntactic sugar for:
let fun quantify () =
  if expression''
    then if expression''
      then (expression; quantify ())
      else quantify ()
    else ()
  end
  in
  quantify ()
end

Notice that when expression'' is left out (it does really serve no special purpose, but is there for orthogonality),
this is exactly the same as the classical while construct:

while expression''' do expression

2.1.2 Set and Map Patterns

To help in writing programs with sets, some standard patterns decomposing sets are provided in HimML.

First, the empty set notation may serve as a pattern, which matches only the empty set. Thus:

case s of
  {} => f()
| _ => g(s)

executes f() when s is the empty set, and g(s) if s is not empty.

A pattern of the form \{p \Rightarrow q\} matches exactly those maps (or sets) of the form \{x \Rightarrow y\}, where p
matches x and q matches y (q and y are assumed to be () if not written). So:

case s of
  {} => f()
| {x \Rightarrow y} => h(x,y)
| _ => g(s)

executes f() when s is the empty set, h(x,y) if s is the map \{x \Rightarrow y\}, and g(s) if s contains at least two
elements.

The \(\cup\) infix function computes the union of two sets. To mimic this behavior, the \(\cup\) infix pattern operator splits a
map (and not only a set) in two disjoint parts, the first one being matched by the pattern on the left of the \(\cup\), the second
one being matched by the pattern on the right. For instance:

\[
fn \{x \Rightarrow y\} \cup rest => g(x,y,rest)
\]

is a function that decomposes its argument by extracting \(x\) from its domain, letting \(y\) be the value that \(x\) is mapped
to, and rest being the argument with \(x\) removed. It then applies \(g\) on the tuple \((x,y,rest)\).

Note that the \(\cup\) may be computationally non trivial. It may be required from \(\cup\) to backtrack to find a match. For
instance:

\[
fn \{\text{ref } x \Rightarrow y\} \cup rest => g(x,y,rest)
\]

is equivalent to:

\[
fn s => case (some (x,y) | ref x => y in map s end) of
  SOME (x,y) =>
    let val rest = (x) <-| s
    in
    g(x,y,rest)
  end
| NONE => raise Match
\]
is the map application function, $\cap$ is the infix “domain restrict by” function.

Other map patterns are expressible as syntactic sugar for patterns using $\cup$:

$$\{ p_1 [ => q_1] ( \cap p_i [ => q_i])^* \} \left[ \text{tt} \cup r \right]$$

is equivalent to:

$$\{ p_1 [ => q_1] \} (\cup \{ p_i [ => q_i] \})^*[ \text{tt} \cup r \right]$$

which is not ambiguous, as $\cup$ associates to the right. Notice that, because of this translation, the pattern $(x, y)$ matches sets of cardinal exactly 2 ($x$ may never be equal to $y$), for example.

The pattern on the left of a $\cup$ is actually constrained to be of the form $\{ p_1 [ => q_1] ( \cap p_i [ => q_i])^* \}$. More general patterns could have been considered, but would have led to a much more complicated evaluator, and were not deemed worth the trouble.

Another map pattern construction generalizes the ellipsis construct already found in Standard ML record and tuple patterns:

$$\{ p_1 [ => q_1] ( \cap p_i [ => q_i])^*, \ldots \}$$

is equivalent to:

$$\{ p_1 [ => q_1] ( \cap p_i [ => q_i])^* \} \cup _\text{tt}$$

A case not covered by the above definition, but which is a natural extension, is the pattern $(\ldots)$ which matches all maps, but no other object.

In general, a pattern of the form $\{ x [ => y] \}$, where $x [ => y]$ does not match all maplets inside the value to be matched, may be slow. Indeed, in this case, the evaluator may have to backtrack to find the correct match. A particularly simple case is the case when $x$ is a constant pattern, like in $\{ "abc" => y \} \cup \text{rest}$. In this situation, the evaluator is optimized so as not to sweep through the map in argument, but to find directly the element mapped to the constant $\text{"abc"}$, match it with the pattern $y$, and match the rest of the map with the pattern $\text{rest}$.

Note that map patterns never get called repetitively, they only return the first correct match. Hence, for example:

$$\{ x | \{ x, \ldots \} \text{ in } s \}$$

where $s$ is a set of sets, does not compute the distributed union of elements of $s$, but a set containing one element taken in each non empty element of $s$.

Finally, the system order may be defined with map patterns like this:

$$\text{fun system_less (x,y) = let val \{ x', \ldots \} = \{ x, y \} \text{ in } x'<>y \text{ end}$$

which works for any pair of objects of the same equality type (an equality type is a type that admits equality). Beware that $x$ may be less than $y$ in the system order during one session and greater than $y$ in another session. This may be a problem when transmitting data from a HimML process to another, or when reading back data that was saved from a file. However, it is a convenient way to get a total order on objects of a given equality type, when the real nature of this order does not matter.

### 2.1.3 Typing Sets and Maps

In the following, the expressions $e_i$ are assumed of type $\tau$, and $e'_i$ are of type $\tau'$:

- $\{ \}$ is of type $\forall \forall' a, \forall' b \cdot a \; \neg m > 'b$ (the empty map is a map of any map type).
- $\{ e_1 => e'_1, \ldots, e_n => e'_n \}$ and $<\{ e_1 => e'_1, \ldots, e_n => e'_n \}$ are of type $\tau \; \neg m > \tau'$.
- $\{ e_1, \ldots, e_n \}$ and $<\{ e_1, \ldots, e_n \}$ are of type $\tau \; \neg m > \text{unit}$, that is $\tau \; \text{set}$.  

11
• The same rules hold for map patterns without the \[U\] construct. If \(p : \tau \rightarrow \tau'\), and \(p' : \tau \rightarrow \tau'\), then \(p \cup p' : \tau \rightarrow \tau'\).

• A domain descriptor \(p\) in list \(e\) is well-typed if \(p : \tau\) and \(e : \tau\) list for some type \(\tau\).

• A domain descriptor \(p [\Rightarrow p']\) in map \(e\) (where \(p'\) is taken to be () is absent) is well-typed if \(p : \tau\), \(p : \tau'\) and \(e : \tau \rightarrow \tau'\) for some types \(\tau\) and \(\tau'\).

• A domain descriptor \(p\) in set \(e\) is well-typed if \(p : \tau\) and \(e : \tau \rightarrow \tau'\) for some types \(\tau\) and \(\tau'\).

• A domain descriptor \(p\) sub map \(e\) is well-typed if \(p : \tau \rightarrow \tau'\) and \(e : \tau \rightarrow \tau'\) for some types \(\tau\) and \(\tau'\).

• A such that \(e\) filter is well-typed if \(e : \text{bool}\).

• In a type context where the pattern variables inside the domain descriptors make the part after the \(|\) or the \(||\) well-typed, if \(e : \tau\) and \(e' : \tau'\) (unit when \(e'\) is absent), then:
  
  - \{\(e[\Rightarrow e'][||]|...\) : \(\tau\rightarrow\tau'\)
  - \(<\{e[\Rightarrow e'][||]|...\) : \(\tau\rightarrow\tau'\)
  - \([e||...\) : \(\tau\) list
  - some \(e||...\) end : \(\tau\) option
  - iterate \(e||...\) end : \(\text{unit}\)

  If \(e : \text{bool}\):
  
  - all \(e||...\) end : \(\text{bool}\)
  - exists \(e||...\) end : \(\text{bool}\)

• If \(e : \tau, e' : \tau'\) (unit when \(e'\) is not present), \(e'' : \text{bool}, e''' : \text{bool}\):
  
  - \{\(e[\Rightarrow e'][|\text{while } e'''[\text{ such that } e'']|]\) : \(\tau\rightarrow\tau'\)
  - \(<\{e[\Rightarrow e'][|\text{while } e'''[\text{ such that } e'']|]\) : \(\tau\rightarrow\tau'\)
  - \([e|\text{while } e'''[\text{ such that } e'']|\) : \(\tau\) list
  - all \(e|\text{while } e'''[\text{ such that } e'']|\) end : \(\text{bool}\)
  - exists \(e|\text{while } e'''[\text{ such that } e'']|\) end : \(\text{bool}\)
  - some \(e|\text{while } e'''[\text{ such that } e'']|\) end : \(\tau\) option
  - iterate \(e|\text{while } e'''[\text{ such that } e'']|\) end : \(\text{unit}\)

These types are deducible from the equivalent forms given in the previous section.

### 2.2 Numbers and Physical Units

Apart from integers, there is only one numerical object in HimML: the complex number with real and imaginary parts represented as floating-point numbers (as \(\text{C doubles}\), i.e. on 64 bits usually). \(1.0, -2.5, 1.0\text{E5}\) and \(1.0 : -2.0\) are all numbers in HimML, their values in standard mathematical notation are \(1, -2.5, 10^5\) and \(1 + 2i\). From version 1.0.11 on, there are also integers, but we shall not call them numbers to avoid confusing matters, and we shall consider the latter rather as counters or sets of bit-fields. \(1, 3\) are integers, not floating-point numbers.

To help programmers write scientific code, typically to model computations on physical dimensions, a system of typing with physical units has been created in HimML (see [5]).

We conceive a \textit{numerical type} as any type that represents a physical quantity, whatever it may be. The simplest numerical type is \text{num}, which is the type of numbers without a unit (all numbers above have type \text{num}). We shall see
that numerical types are monomials over a set of basic dimensions (basic numerical types, like mass, speed, etc.), that can be declared with the dimension keyword. Units or scales are scale factors inside a dimension (kilogram, ton or ounce are units, or scales of the dimension called mass).

A numerical type variable can be instantiated only by a numerical type. It is written with a sharp (#) sign in the name, following all quotes, underscores and digits. For instance, ‘#a is a numerical type variable, ‘#a is a numerical type variable with the equality attribute (this is futile: all numerical types admit equality), ‘#1#a is an imperative numerical type variable of strength 1.

The declaration: dimension mass(kg) declares a new base dimension, called mass, which is incomparable with all other base dimensions. As it is a declaration, it obeys all the usual lexical scoping rules of declarations like val, fun, type... It also declares a scale by default, called kg (this scale is optional; if it is not provided, as in dimension person, then the default scale has the same name as the dimension). Scale names can be appended to numbers to indicate their type: typing 1'kg; at the toplevel after the above declaration will result in the display:

```
  it : mass
  it = 1 'kg
```

Numerical types can be multiplied or raised to a power (actually, the definition of a numerical type is any product of powers of dimensions and numerical type variables). For any numerical types t1 and t2, t1*t2 is the product of both types, and t1^n is the power of t1 to the number n. For instance, we can write:

```
  dimension distance(m) and time(s);
  type speed = distance'time^ ˜1;
  type acceleration = distance'time^ ˜2;
```

Scales too can be multiplied and raised to a power. We may declare new scales by multiplying and raising old ones to a power, with the scale declaration (obeying the lexical scoping rules of declarations):

```
  scale 1 'km = 1000 'm; (* 1 kilometer is 1000 meters *)
  scale 1 'h = 3600 's; (* 1 hour is 2600 seconds *)
  scale 1 'kph = 1 'km'h^ ˜1; (* kilometer per hours *)
```

Scale declarations interfere with syntax analysis, so now 130 'km'h^ ˜1 is understood by the HimML toplevel, which answers:

```
  it : distance'time^ ˜1
  it = 36.111111111111111 'm's^ ˜1
```

Thus HimML manipulates not only numbers but numerical values, that is pairs of numbers with a scale. Though, scaling is done during elaboration (the typing phase), so the evaluator never bothers with different scales and is therefore as fast as if there were no scaling facility.

Numerical typing is also fully polymorphic, thanks to numerical type variables, and the typing algorithm infers most general types, even with numerical types (the notion of most general must however be relativized). For instance, addition is + : ‘#a * ‘#a -> ‘#a. That is, it takes two numerical values representing the same physical dimension, and returns a value of the same type. This effectively prohibits adding distances with energies for instance, while allowing programmers to declare measurements they want to add. More complex types can be used; for instance, division is / : ‘#a * ‘#b -> ‘#a ' ‘#b^ ˜1, which means that it takes two numerical values having independent types, and returns a value having the quotient type.

Transcendental functions like log or special functions like floor cannot have fully polymorphic numerical types, because it would make no sense. This is why the type of log is num -> num, as the type of floor. Note that type conversions are possible: for example, if x is of type mass, we can convert it to a distance by writing x*1'm'kg^ ˜1.

A special case for which we have to be careful is the power operation **. Indeed, ** is basically a transcendental function, calling exp and log, so is of type num * num -> num. But this type is too restrictive: writing x**2 would imply that x ought to be of type num. To alleviate this, the typing algorithm knows the special case when the second argument to ** is an explicit numerical constant n (with no scale): the operation fn x => x**n is then of type ‘#a -> ‘#a^n.

Some other subtleties worth mentioning are the following. First, all dimensionless constants (like 1.0, ˜3.5) have type num, except 0.0, whose type is ‘#a; in other words, 0.0 has all numerical types. This is consistent with the semantics of units. However, 0'kg still has only the mass type, for example. Notice, by the way, that we write
dimensionless constants with a dot and a zero figure after the dot (like $0.0$ instead of 0). This is to disambiguate between the number 0.0 (of type num) and the integer 0 (of type int); if you put scales after the constant, this is not needed since there is no ambiguity.

Then, it is possible to declare scales with negative or even complex scale factors. For example, the declaration
\[ \text{scale } 1'a = -2'b \]

is legal, provided \( b \) is an already declared scale. This poses a problem, in that comparison functions like \(<\) then seem to be ill-defined: indeed, assume \( 1'b > 0 \) holds, then as \( -0.5'a = 1'b \), we should have \( -0.5'a > 0 \), which is odd. The solution is to say that all comparisons (as well as all other functions on numerical values) are done with respect to the default scale. So, in the example, if \( b \) is the default scale, then \( 1'b > 0 \) and \( -0.5'a > 0 \).

For a list of dimensions and scales, look at the file ‘units.ml’. Note that units can be used not only to represent physical dimensions, but also multiplier prefixes (scale \( 1'k = 1000.0 \), so that \( 1'km = 1'k'm \)), or more abstract dimensions (dimension memory(byte), dimension apples and oranges, etc.).

### 2.3 Other Differences

Here we list other differences between HimML and Standard ML.

- As \{ and \} are the most natural candidates to delimit sets and maps, they are used to this purpose in HimML. Consequently, they cannot be used as record delimiters as in Standard ML. Instead, record expressions, record patterns and record types are delimited with \[ [\] \] as in ML.

- map types, and some others too (dynamics, promises, continuations, numerical types) have been added to ML: see section 3.

- on the chapter of types, contrarily to Standard ML where type functions are total, in HimML they are partial. Indeed, whereas in Standard ML any type constructor may be applied to any type, this is not so in HimML. The most prominent cases are operations on numerical types. For example, if we write:

  type ‘a pair = ‘a * ‘a;
  type ‘’a eqpair = ‘’a * ‘’a;
  type ‘a wrong_square = ‘aˆ2;
  type ‘#a right_square = ‘#aˆ2;
  type (‘a,’b) wrong_relation = (‘a * ‘b) set;
  type (‘’a,’’b) right_relation = (‘’a * ‘’b) set;

then pair and eqpair denote different types: the latter can only be applied on types that admit equality (to simplify, types not built with the function arrow, nor with promises, dynamics or continuations); in Standard ML, the notational difference between ‘a and ‘’a would have been ignored. wrong_square is illegal in HimML because you cannot take the square of the non-numeric type ‘a; on the other hand, right_square is correct because ‘#a is explicitly restricted to denote only numerical types, and is then a type function whose domain is that of numerical types. The same holds of wrong_relation and right_relation, this time considering types that admit equality instead of numerical types (sets can only be built with values that can be compared by the equality predicate).

- In Standard ML, a tuple is a record whose field names are numerical. Not so in HimML: tuples are different from records. The reason is that we allow extensible tuple and record types, that is, types whose length or whose set of fields is not completely determined, and that extension of tuples and of records do not have the same semantics. Actually, extension of tuples is a single inheritance mechanism, whereas extension of records is a multiple inheritance device (see section 3). One of the consequences is that numerical labels are forbidden in records. Another is that () is the empty tuple, of type unit, but that [||] is the empty record, of type [||]. Yet another is that, though there are records with only one field, there are no 1-tuples in HimML, because there would be no way of coding one; indeed, the notation (e) does not represented the 1-tuple built on e, but e directly.

14
• In Standard ML, the only operations available on records are construction (\(|[\text{label}_i = \text{value}_i, 1 \leq i \leq n]|\)), and field selection (\(\#\text{label}\)). In particular, building a record with some fields changed is monomorphic: it is impossible to change fields \(a\) and \(b\) in a record \(r\), say, without rebuilding the whole record as \(|[a = x, b = y, ...]|\). HimML provides the ++\(|[\dots]|\) construction for this purpose. In the case above, we could write \(r++|[a=x,b=y]|\). This is mostly a notational convenience, as it still rebuilds the whole record at run-time. But it also allows one to write code as above, regardless of which fields are present or not in \(r\); so, if new fields are added to the type of \(r\) in a later version of the same software, code written this way won’t break.

• The abstype keyword, in Standard ML, introduces abstract data types, that is, data types whose constructors are hidden, and which are accessible only through a set of functions, defined by the programmer. To make the implementation of these data types completely hidden from the outside, the designers of Standard ML have chosen to hide the equality attributes of these types. That is, in Standard ML, no abstract type admits equality. We felt that it was too bad, and allowed the programmer to explicitly state that he wanted the equality attribute exported, through an eqtype declaration.

• As in Standard ML of New Jersey, and contrarily to Standard ML, nested recs in val declarations are forbidden. Hence, \(\text{val rec f = ... and rec g = ... and val rec rec f = ...}\) are not parsed in HimML.

• The names of exceptions are entirely different in HimML and Standard ML, except for \(\text{Match}\) and \(\text{Bind}\). Some have been added (Empty, ParSweep, NoMem) to accommodate the HimML extensions to Standard ML, others have been eliminated (arithmetic exceptions mainly), because of their poor practical value, already recognized in Standard ML of New Jersey. The exceptions associated with file input/output have also been completely redesigned, for file handling is dealt with completely differently in HimML and Standard ML (see section 4.9).

• Due to the presence of maps in the language, it is very easy to code memoizing functions (functions that remember the value they compute on previous arguments). However, Standard ML’s type system would give \(\_a \rightarrow \_b\) as the most general type for memoizing functions, instead of \(\_a \rightarrow \_b\). To alleviate this problem, and also to simplify the coding of memo functions, keywords memofn and memofun have been added, that are the memoizing analogues of fn and fun.

• Finally, some details: parenthesized type variable sequences with only one variable inside are allowed, e.g.

\[
\text{type ('a) foo = 'a bar}
\]

is allowed. The if, then, and else constructs are not derived forms, so they don’t change meanings when redefining true or false. Similarly, the list expressions written in brackets don’t change meanings when :: is redefined. Type variables may begin with one or two primes, but no more (’’a is forbidden). Finally, type variables that could not be generalized at toplevel do not give rise to an error, but only to a warning, and will be bound to actual types as soon as context permits.
Chapter 3

Core Types

- A type variable is an identifier beginning with a quote ‘. Following the quote, there may be:
  - an optional quote, signalling that this is an equality type variable, that can be instantiated only with equality types. Intuitively, equality types are the types of those objects that may be safely tested for equality. This excludes essentially functions, promises, dynamics, continuations and abstract types with no specified equality, and all types built from such types, except for ref and array types.
  - next, an optional integer, greater than or equal to 1. The variable is then a weak type variable (in Standard ML of New Jersey parlance, where they were introduced), or an imperative type variable (in Standard ML parlance). The integer is called the strength of the type variable. Non weak type variables may be thought as having strength infinity. The strength of a type expression is the minimum of the strengths of its type variables (infinity if it has none). A weak type variable may only be instantiated by a type of equal or less strength. The strength of a weak type variable in the type of an expression \( f \) may be thought as the minimum number \( n \) of arguments \( e_1, \ldots, e_n \) such that evaluating \( f(e_1) \ldots (e_n) \) creates the corresponding mutable object (ref or array in general).
  - next, an optional sharp (#) sign. The variable is then a numerical type variable, which may be instantiated only with numerical types. A numerical type is either a numerical type variable, the special type num, the name of a dimension (as defined by a dimension declaration), a numerical product \( \tau \cdot \tau' \) of numerical types, or a numerical power of a numerical type \( \tau^x \), where \( x \) is a number (note that if the number has a negative real part, there should be a blank between the ‘ and the ‘, because ‘’ is a valid ML symbol; however, in this context the parser is smart enough to know that such a symbol would be useless, and correctly recognizes ‘’ as though it were ‘’).
    Any numerical type already admits equality, so the extra quote indicating an equality type variable is superfluous in the case of numerical type variables.
  - finally, a sequence of letters, digits, quotes or underscores beginning with a letter.

Contrarily to Standard ML, attributes of type variables matter in type declarations. For example:

\[
\text{type } ' ' a \text{ pair } = ' ' a \times ' ' a
\]

declares a polymorphic type of pairs, but this type is restricted to pairs of values admitting equality. The same declaration in Standard ML would incur no such restriction. This feature is necessary in HimML because type functions are partial, whereas they are total (they apply to all types) in Standard ML. Indeed, the numerical

---

1In Standard ML, no abstract type admits equality; this restriction is lifted in HimML.
product and the numerical power of types apply only to numerical types, and the map type constructor may only be applied to two types, the first of which having to admit equality.

Therefore, type declarations such as:

```ocaml
type 'a product = 'a * 'a
type 'a *'b map = 'a -> 'b
```

would have no meaning in HimML. We should have written:

```ocaml
type '#a product = '#a * '#a
type 'a *'b map = 'a -> 'b
```

The imperative nature of type variables is not checked, however, in conformance with the Definition of Standard ML. Therefore, declaring:

```ocaml
type '1a fakearray = '1a ref list
```

does not preclude the use of the type expression `'2a fakearray, though it seems to break the rule of not replacing a type variable by a type of higher strength, and is in fact mostly equivalent to:

```ocaml
type 'a fakearray = 'a ref list
```

The only difference is with datatypes, where:

```ocaml
datatype 'a fakearray = fake of 'a ref list
```

builds a constructor `fake : 'a ref list -> 'a fakearray, whereas declaring

```ocaml
datatype '1a fakearray = fake of '1a ref list
```

would declare the constructor with the more restrictive type `fake : '1a ref list -> 'a fakearray.

- **string** is the type of all character strings. There is no character type in HimML. However, all one-character strings are pre-allocated, and play the role of characters. There is no limit on the length of strings except the length of the largest free block in memory.

- **int** is the type of integers. Integers are machine integers only, for now. They will be replaced by arbitrary precision integers (bignums) in a future version, but for now all overflows merely give rise to undefined behavior, contrarily to what the definition of Standard ML prescribes (such overflows should raise exceptions). Integers are considered as considered as counters or as bit-fields, but not as proper numbers, which are represented by objects of numerical types.
  Machine integers are in the range `[min_int, max_int]`, and are represented with `nbits` bits.

- **num** is the type of all numbers, that is, numerical values with no dimension. Numerical values are stored as complex numbers with real and imaginary parts in floating point format, double precision (64 bits).
  There is no **real** type as in Standard ML. All Standard ML functions using **real** arguments, or producing **real** arguments use or produce numerical values in HimML, whether of type **num** or of a more general numerical type.

- **bool** is the datatype of boolean values:
datatype bool = false | true

Note that false and true are datatype constructors, and thus may be used in patterns to match themselves. Contrarily to Standard ML, while loops, if conditionals are not defined as syntactic sugar, and so don’t change their behavior under a redefinition of true and false. This is also valid of all new comprehension, quantification and iteration constructs provided in HimML.

- list is the predefined datatype of lists. It is declared by:

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

  :: is furthermore declared as having infix status, being right associative, and having precedence 5. This is as though we had typed:

  infixr 5 ::

- option is the predefined datatype of optional values. It is declared by:

  datatype 'a option = NONE | SOME of 'a

  NONE represents the absence of value, and SOME x represents the presence of the value x.

- void is a predefined empty datatype. It is declared by:

  datatype void = VOID of void

  and has as sole constructor VOID : void -> void. As you may check, it is impossible to construct a ML value of type void; this is why it is a void type.

  Stupid as it may seem, it is useful at least in the following case. Assume you want to provide a function quit that never returns. Because quit never returns, it is likely that its type will be of the form τ -> 'a, where ‘a is a new type variable. Now, you may want to declare the type of such functions that never return, as follows:

  type never_returns : τ -> 'a

  where τ is the argument type, and ‘a is a new type variable, making it clear that such a function indeed cannot return. Unfortunately, you cannot write this in ML, because ‘a should be given as a parameter to the type constructor never_returns, i.e., it forces you to write type ‘a never_returns = ..., which is probably not what you want. The void is a workaround:

  type never_returns : τ -> void

  indeed states that a function of type never_returns can only return an object of type void, namely something that does not exist. On the other hand, if f is a function of this type, expressions like

  if ... then f() else g()

  won’t type-check if g() is an expression that returns a result; the solution is to change f() into, say, the sequence (f(); raise NonSense), which now has any type, and where exception NonSense cannot be raised.

- ref is the pseudo-datatype of polymorphic mutable references. You may think of it as declared with:

  ref
datatyp 'la ref = ref of 'la

which also declares a constructor ref : 'la -> 'la ref.

* array is the type of polymorphic mutable arrays. They were introduced in Standard ML of New Jersey, and are somehow a generalization of refs, with multiple indexed mutable entries.

* intarray is the type of (non-polymorphic) arrays of integers. Integer arrays take less space than objects of type int array and put less pressure on the garbage collector, but are less flexible in that they can only contain objects of type int.

* exn is the extensible pseudo-datatype of exceptions. Exceptions are declared with the exception declaration, and raised with the raise construct and caught with the handle construct.

* promise is the abstract type of promises. Promises are a poor man’s implementation of lazy data structures. When e is an HimML expression of type τ, the expression delay e builds a promise, that is a structure representing the future evaluation of e, of type τpromise. Evaluation of e if forced with the force : 'a promise -> 'a function. An expression inside a promise is only evaluated once: forcing a promise stores the result in place of e inside the promise, and sets a flag indicating that the promise has been forced.

Considering that HimML has memoizing functions, delay e is roughly equivalent to memofn () => e; then force is fn p => p(), with 'a promise = unit -> 'a. However, promises are more space-efficient than memoizing functions, and delay and force are more explicit.

Promises come from the algorithmic language Scheme [12]. Promises do not admit equality.

* cont is the abstract type of continuations, inspired from Scheme [12], and typed as in [2]. A continuation is reified with the callcc function (or the catch function) and thrown with the throw function.

* dynamic is the special type of dynamic values. Dynamic values may be thought as pairs (ν, σ), where e is a value and σ is a type, such that e if of type σ. Dynamics are built with the pack pseudo-datatype constructor.

Dynamics are the basis for functions that must operate on data of different types, but do something depending on the actual type of the data. A different sort of dynamics was introduced in [1], ours is more in the spirit of Mycroft (cited in the paper). It corresponds rather precisely to the dynamic construct (with type dyn) of CAML [7]. Uses are type-safe communications of data between independent processes, checkpointing processes (saving data in a file to retrieve them later), polymorphic printing functions, and so on.

dynamic does not admit equality, because this would not make sense in the general case.

* unit is the type of the 0-ary tuple (). Contrarily to Standard ML, this is not the same as the empty record | | (noted () in Standard ML; this notation was abandoned in HimML because it was incompatible with the map notation).

* * is the infix n-ary type product constructor (n ≥ 2): τ₁ * ... * τₙ is the type of all tuples (e₁,...,eₙ) with e₁ : τ₁, ..., eₙ : τₙ.

As an extension to the Standard ML type system, extensible tuple types are introduced in HimML:

\[ τ₁ * ... * τₙ * ... : \text{rest} \]

is the type of all tuples (e₁,...,eₘ) with m ≥ n and e₁ : τ₁,...,eₙ : τₙ (an exception is the case n = 1, where it is not the type of any 1-ary tuple, only 2-ary and higher tuples, because there are just no such things as 1-ary tuples in HimML).

This allows the HimML type system to lift some restrictions on the typing of tuple patterns with an ellipsis: the pattern \( (p₁,...,p_n,...) \) has type τ₁ * ... * τₙ * ... : 'a, if pᵢ : τᵢ for all i. As a special case, the selector function #n (n ≥ 1) is fully polymorphic in HimML (not in Standard ML), and has type:
The type qualifying the ellipsis must be a type variable. This type variable can only be substituted for by a tuple type. If it admits equality (say, \texttt{' \textsf{r} \textsf{est} \textsf{}}), then all underscribed components are to be of equality types; imperative and weak type variables (say, \texttt{' \textsf{u} \textsf{rest} \textsf{}} with \( n \geq 1 \)), constrains all underscribed components to be of imperative types with strengths at least \( n \); Combinations are possible, too.

A tuple type admits equality if and only if all its component types admit equality (including its possible ellipsis). The strength of a tuple type is the minimum strength of its component types (and of its ellipsis, if present).

Extension of extensible tuple types is done by instantiating the extension variable. However, extensions are mostly built by the type inferencing engine. Single inheritance is automatic. To take an example, the selector function \texttt{\#l} has type \texttt{'a * ... *'rest \textsf{}} -> \texttt{'a}. When applied to, say,

\begin{verbatim}
(1,"abc",false) : num * string * bool
\end{verbatim}

the type of \texttt{\#l} is automatically specialized to become \texttt{num * string * bool \textsf{}} -> num.

- For any \( n \) types \( \tau_1, \ldots, \tau_n \), and any distinct label identifiers \( \texttt{lab}_1, \ldots, \texttt{lab}_n \) (\( n \geq 0 \)),

\begin{verbatim}
| [\texttt{lab}_1 : \tau_1, \ldots, \texttt{lab}_n : \tau_n] |
\end{verbatim}

is the type of all records having exactly the \( \texttt{lab}_i \) as field labels, the field \( \texttt{lab}_i \) having type \( \tau_i \). Labels are ordinary ML identifiers, that is sequences of letters, digits, quotes or underscores beginning with a letter; contrarily to Standard ML, positive integers (so-called numerical labels;numerical;) are not considered labels. In particular, tuples are not special cases of records. Moreover, the delimiting characters for records and record types in HimML are \texttt{|[ and ]|} instead of \texttt{|{ and }|}; this choice was made so as not to conflict with the map constructions of HimML.

As for tuples, extensible record types are provided: \texttt{| [\texttt{lab}_1 : \tau_1, \ldots, \texttt{lab}_n : \tau_n, \ldots : 'rest] |} is the type of all records having at least the \( \texttt{lab}_i \) as labels, the field \( \texttt{lab}_i \) having type \( \tau_i \). The extension variable \texttt{'rest} may be constrained to admit equality, or to be imperative or weak, as for tuple extensions. This type variable can only be substituted for by a record type.

This enables full typing of record patterns with ellipsis, which Standard ML restricts because of its less powerful type system. This extension of Standard ML typing is a weaker extension than the one proposed in [13].

As in Standard ML, the field selector \texttt{\#lab} is the function that picks the field \( \texttt{lab} \) out of the record passed as an argument. Contrarily to Standard ML, it is fully typable:

\begin{verbatim}
\#lab : | [\texttt{lab} : 'a, ... : 'rest] | -> 'a
\end{verbatim}

Contrarily to Standard ML again, records are entirely different from tuples. In Standard ML, tuples are just records with only numerical labels. In HimML, record labels cannot be numerical. This was dictated by a choice of implementation of records entirely different from the implementation of tuples. Witness the semantic difference between ellipsis in tuple and record types: were tuples to be special cases of records, an extensible tuple type like \texttt{a \* b \* ... : 'r} would actually stand for \texttt{|[1 : 'a, 2 : 'b, ... : 'r]|}, which would then contain the type \texttt{|[1 : 'a, 2 : 'b,extra : 'c]|}, which is not the type of any tuple, because \texttt{extra} is not an integer. We want to consider extensible tuple types as containing only tuple types.

A record type admits equality if and only if all its field types admit equality (including those subsumed by its ellipsis, if any). The strength of a record type is the minimum strength of its field types (and of its ellipsis, if present).

Extension of extensible record types is done by instantiating the extension variable. However, extensions are mostly built by the type inferencing engine. Multiple inheritance is automatic: if \( (\alpha_1^1, \ldots, \alpha_{n_1}^1, \rho_1^1) \tau_1, \ldots, (\alpha_1^n, \ldots, \alpha_{n_m}^n, \rho^m) \tau_n \) are all extensible record types (where by convention, the extension variable is \( \rho^l \), the last variable), and \( \tau \) is a record type (extensible or not), the construct:

\begin{verbatim}
20
is a new record type, having all the fields of the types $\tau_1, \ldots, \tau_n, \tau'$, and which is extensible if and only if $\tau'$ is extensible. The resulting labels are associated with the types they have in the $\tau_i$ and $\tau'$. If two types provide the same label, they must associate it with the same type. Notice also that the types involved may not be records, but more complicated types involving extensible record types.

We get automatic multiple inheritance on records with this scheme, thanks to type inference. For example, if $(lab)$ is an identifier, the selector $(\#lab)$ is the function that takes as argument any record with a label $lab$, and returns the corresponding field. Thus, $(\#lab)$ has type $\tau \to \tau'$ in HimML (it is not typeable in Standard ML). Then, a function such as:

$$\text{fn data => \#multiplicand(data) * \#multiplier(data)}$$

has the effect that the variable $data$ inherits the types $\tau_i$ and $\tau'$.

- $\to$ is the infix function type constructor. All functions in ML are basically unary. Zero-ary functions may be simulated as functions taking () as argument. n-ary functions ($n \geq 2$) are functions taking n-tuples as arguments. Variable arity functions may be coded as functions taking list arguments. If arguments to a function must be referred to with keywords, it is possible to use record arguments, where the record labels serve as keywords. If a n-ary function does not depend on the order or multiplicity of their arguments, it is natural to encode the arguments as a set (for the logical ‘or’ or ‘and’ on formulas, for an equality function, etc.).

- ‘$\hat{}$’ is the numerical multiplication type constructor. If $\tau$ and $\tau'$ are numerical types, then $\tau \hat{} \tau'$ is the numerical type of the product of any two values of types $\tau$ and $\tau'$. For example, if we have $\text{dimension intensity(A)}$ and $\text{type area = distance}^{\hat{}2}$.

The ‘$\hat{}$’ symbol is also used as the scale multiplication operator: the value of $3 \hat{} \tau \times 4 \hat{} \tau$ is $12 \hat{} \tau \times \tau$.

The ‘$\hat{}$’ symbol is also used as the power operator for scales; e.g., the value of $1/100 \ s$ is $0.01 \ s^{\hat{} -1}$. There can be no confusion between the two uses of ‘$\hat{}$’, as with the ‘$\hat{}$’ in infix string concatenation operator.

- $\hat{}$ is the numerical power type constructor. If $n$ is any number (that is, any complex number, without dimension), and $\tau$ is a numerical type, then $\tau^{\hat{}n}$ is a numerical type; e.g., we may write $\text{dimension distance(m)}$ and $\text{type area = distance}^{\hat{}2}$.

The ‘$\hat{}$’ symbol is also used as the power operator for scales; e.g., the value of $1/100 \ s$ is $0.01 \ s^{\hat{} -1}$. There can be no confusion between the two uses of ‘$\hat{}$’, as with the ‘$\hat{}$’ in infix string concatenation operator.

- $\hat{}$ is the infixed map type constructor. If $\tau$ and $\tau'$ are types, and on the condition that $\tau$ admits equality, then $\tau \rightarrow \tau'$ is the type of maps mapping objects of type $\tau$ to objects of type $\tau'$. The resulting map type admits equality whenever $\tau'$ admits equality.

When $\tau'$ is unit, $\tau \rightarrow \tau'$ is written $\tau$ set, and represents the types of finite sets of objects of type $\tau$.

- polymorphic datatypes and abstract datatypes are provided exactly as in Standard ML. An abstract datatype declaration has the form:

$$\text{abstype implementation with interface end}$$
where *implementation* is a standard datatype binding list (what usually follows a `datatype` keyword), and *interface* is a list of declarations.

In Standard ML, no declared abstract datatype admits equality, to preserve its abstract character. In HimML, however, equality declarations may be used inside the *interface* part, to define the equality predicate on an abstract datatype. Equality declarations take the form:

```
eqtype τ₁ [and τᵢ]^
```

where the $τᵢ$ are amongst the declared abstypes. This declaration must be the first in the interface. It is an error to declare a $τᵢ$ as admitting equality with this declaration, if it did not admit equality according to the rules of equality of datatype bindings (see [6] and [9]).
Chapter 4

The Standard Library

4.1 General Purpose Exceptions and Functions

The following exceptions are defined:

- **exception Bind** is raised when a value binding (val or val rec) fails to match. The Definition of standard ML stipulates that Bind may be raised only in situations where a match is not exhaustive, and that the compiler must warn the user of this. Map pattern matching complicates a lot static analysis of pattern matching, so the HimML compiler does not warn the user for now. HimML does not give any warning about redundancy of patterns either.

  A typical example of a non-exhaustive pattern is:

  ```ml
  let val a::l = r in a end
  ```

  which raises Bind if r is the empty list.

- **exception Match** is raised when a function cannot be applied to its argument, though they agree on types. A typical example is:

  ```ml
  (fn (a::l) => a) r
  ```

  or the equivalent:

  ```ml
  case r of a::l => a
  ```

  which raises Match if r is the empty list.

- **exception ParSweep** is raised when a comprehension (or quantification, or iteration) using the || separator is meant to sweep through domains of different cardinalities (the $n_i$ in the explanation of comprehensions in section 2.1.1).

  However, the moment where this exception is raised is voluntarily left unspecified. For example, the current interpreter checks the cardinalities before sweeping, but it would be easier to check them after sweeping in compiled code, for example.

- **exception NoMem** is raised when the garbage collector fails to get enough space to complete a computation. This exception cannot be handled in production code, because trying to correct the problem usually implies that we allocate some memory, which only makes the situation worse.
• exception Stack was intended to be raised whenever the stack overflowed. However, in HimML, it never overflows, because when the stack becomes full, the system reifies the current continuation, empties the stack and proceeds with the computation.

• exception NonSense is raised when evaluating a non-sensical expression. In theory, whether it is in Standard ML or in HimML, this can never happen, because all expressions that type-check are well-formed. However, when some expression fails to type-check, it is usually bad practice to stop compiling immediately. The usual solution is to allow for several errors to occur before stopping compilation. In HimML, we have chosen not to stop compilation at all, and instead to compile a special expression \texttt{raise NonSense} in place of all non-sensical (ill-typed) expressions. This exception can also be raised when reloading a module from a file, and then executing some code that contains continuations coming from that file. Continuations cannot be saved to disk, so NonSense is then raised. If this happens, the system should have given a warning when loading the module, saying that it was attempted to load a continuation object. If this happens, it is probably a bug in HimML.

• exception Catch is raised when a continuation captured by \texttt{catch} (not by \texttt{callcc}) is thrown outside of its defining dynamic scope (i.e., thrown twice, including the implicit throw that happens at the end of the evaluation of the body of the function given to \texttt{catch} or \texttt{callcc}). A continuation reified by \texttt{callcc} will never give rise to such a problem, however they are slower to handle. The exception \texttt{Catch} is not systematically raised each time a continuation captured by \texttt{catch} is thrown twice. Sometimes, such continuations may assume the behavior of \texttt{callcc} continuations.

• exception ReturnToTheFuture is an exception that should very rarely be raised. Its purpose is to correct a subtle problem with continuations. At each instant, there is a notion of age in HimML, which counts the number of toplevel declarations that have been evaluated. Continuations record the age where they were created. Then, \texttt{throw} is illegal on future continuations (with an age higher than the current one). If \texttt{throw} is applied on such continuations, the exception \texttt{ReturnToTheFuture} is raised. This can happen because we can create a toplevel reference \texttt{r} pointing to a continuation, capture the current continuation (at age \texttt{n}), then evaluate some declarations, capture the current continuation (at age \texttt{n'} > \texttt{n}), store it in \texttt{r}, then relaunch the old age \texttt{n} continuation. If now, we evaluate some other declarations and then throw the age \texttt{n'} continuation stored in \texttt{r}, we would get into an inconsistent state, because different time lines cannot cohabit in the current implementation (the only way we know to do it would be to slow down the whole implementation drastically). So we forbid this.

General functions are:

• \texttt{identity : 'a -> 'a} is the identity function \texttt{fn x => x}.

• \texttt{not : bool -> bool} is the logical negation, \texttt{fn false => true | true => false}.

• \texttt{=} : ''a * ''a -> bool is the polymorphic equality predicate, defined only on equality types. It is declared infix of precedence 4. It is advised not to change this, as the HimML parser depends on it.

• \texttt{<> : ''a * ''a -> bool} is the polymorphic difference predicate, that is \texttt{fn (x,y) => not (x=y)}. It is declared infix of precedence 4.

• \texttt{o : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)} is the function composition operator. It is declared infix, left associative and of precedence 3. This function could also have been defined as \texttt{fn (f,g) => fn x => f(g(x))}

• \texttt{before : 'a * 'b -> 'a} evaluates its two arguments, and returns the first. It is declared infix, left associative of precedence 0. It is used as in \texttt{e before e'}, which evaluates \texttt{e}, then \texttt{e'}, and returns the result of \texttt{e}.
4.2 Lists

The exception:

```
exception Nth
```

is defined. It is raised whenever it is attempted to access an element outside of a list. Recall that lists are defined by the datatype declaration:

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

:: : 'a * 'a list -> 'a list is the list constructor, and is declared infix, right associative of precedence 5.

- null : 'a list -> bool returns true if the list in argument is empty. This is equivalent to

  ```
  fn nil => true | _ => false
  ```

- hd : 'a list -> 'a returns the first element of the list in argument, or raises Match if the list is empty. This is equivalent to:

  ```
  fun hd (x :: _) = x
  ```

- tl : 'a list -> 'a list returns the list composed of all but the first element of the list in argument, or raises Match if the list is empty. This is equivalent to:

  ```
  fun tl (_ :: x) = x
  ```

- len : 'a list -> int computes the length of a list.

- nth : 'a list * int -> 'a gets the \(n + 1\)th element of the list \(l\) when applied to the arguments \(l\) and \(n\). If \(n\) is out of range \((n < 0\) or \(n >= \text{len}\ l\))\), Nth is raised. Note that elements are indexed starting at 0. nth is declared infix, left associative of precedence 9.

- nthtail : 'a list * int -> 'a list gets the \(n\)th tail of the list \(l\) when applied to the arguments \(l\) and \(n\). If \(n\) out of range \((n < 0\) or \(n >= \text{len}\ l\))\), Nth is raised. Note that tails are indexed starting at 0. When \(n < \text{len}\ l\), op nthtail \((l, n)\) is the first element of \(\text{nthtail}\ l, n\); if \(n = \text{len}\ l\), nthtail \((l, n)\) is nil.

- @ : 'a list * 'a list -> 'a list concatenates two lists. It could have been defined as:

  ```
  fun op @ (nil,l) = l | op @ (a::l1,l2) = a:: op @(l1,l2)
  ```

@ is declared infix, right associative (as in Standard ML of New Jersey; the Definition of Standard ML defines it to be left associative) of precedence 5.

- append : 'a list list -> 'a list is the distributed concatenation of lists. It could be defined as:

  ```
  fun append nil = nil | append (l::r) = l @ append r
  ```

The application of append to a list comprehension is compiled in an optimized form, where the concatenations are done on the fly, without building the list comprehension first.

- revappend : 'a list * 'a list -> 'a list appends the reversion of the first list to the second list. We could define:
fun revappend (nil,l) = l | revappend (a::l1,l2) = revappend (l1,a::l2)

• rev : ‘a list -> ‘a list reverses a list. It could be defined as:

fun rev l = revappend (l,nil)

• map : (‘a -> ‘b) -> ‘a list -> ‘b list applies its first argument to each element in turn of the list in second argument, and return the list of all results. This is equivalent to:

fun map f l = [f x | x in list l]

• app : (‘a -> ‘b) -> ‘a list -> unit applies its first argument to each element in turn of the list in second argument. This is used purely for side-effects.

fun app f l = iterate f x | x in list l end

• fold : (‘a * ‘b -> ‘b) -> ‘b -> ‘a list -> ‘b combines the elements of the list in third argument by applying the binary operation in first argument on all elements of the list. The second argument is assumed to be the neutral element of the binary operation. For example, fold (op +) 0 l computes the sum of the elements of the list l. fold computes from the right end of the list towards the left end. It can be defined by:

fun fold f b = 
  let fun f2 nil = b 
      | f2 (e :: r) = f(e,f2 r) 
  in
    f2
  end

• revfold : (‘a * ‘b -> ‘b) -> ‘b -> ‘a list -> ‘b combines the elements of the list in third argument by applying the binary operation in first argument on all elements of the list. The second argument is assumed to be the neutral element of the binary operation. For example, revfold (op +) 0 l computes the sum of the elements of the list l. revfold computes from the left end of the list towards the right end. It can be defined by:

fun revfold f b = 
  let fun f2 b nil = b 
      | f2 b (e::r) = f2 (f (e,b)) r 
  in
    f2
  end

4.3 Sets and Maps

Exceptions related to sets and maps are:

• exception MapGet, raised when a map is applied to an element outside of its domain (with the ? function).

• exception Empty, raised when taking the distributed intersection of the empty set, or choosing an element in the empty map.
exception Range, raised when trying to build a range \(\{m, \ldots, n\}\) with non-integer bounds (with the to operator).

Functions on sets and maps are:

- **empty** : \(\forall 'a. 'a 
  \rightarrow \text{bool}\) tests whether a map is empty. It is equivalent to \(\lambda m. m={}\).

- **?** : \(\forall 'a. 'a 
  \rightarrow \text{bool}\) applies a map to an element. It is curried, so that the expression \(?m(a)\) retrieves the element associated with \(a\) in \(m\). If there is no element associated with \(a\) in \(m\), the exception MapGet is raised.

- **inset** : \(\forall 'a. 'a \rightarrow \text{bool}\) returns true if its first argument belongs to the domain of the map in second argument. It can be defined as:

  \[
  \text{fun op inset } (a,m) = (?m(a); true) \text{ handle MapGet } \Rightarrow \text{ false}
  \]

  It is declared infix, of precedence 4.

- **inset** and **inset** share a one-entry cache, where the last maplet is stored, so that testing **inset** then using **?** incurs almost no speed penalty.

- **inmap** : \(\forall 'a. 'a 
  \rightarrow \text{bool}\) returns true if its first argument is a pair \((x,y)\) such that the maplet \(x \rightarrow y\) is in the map in second argument. It can be defined as:

  \[
  \text{fun op inmap } (a,m) = (?m(a)=b) \text{ handle MapGet } \Rightarrow \text{ false}
  \]

  It is declared infix, of precedence 4.

- **inset** and **inset** share a one-entry cache, where the last maplet is stored, so that testing **inset** then using **?** incurs almost no speed penalty.

- **subset** : \(\forall 'a. 'a 
  \rightarrow \text{bool}\) tests the inclusion of the domains of maps in argument. It may be defined as:

  \[
  \text{fun op subset } (m1,m2) = \text{ all } x \text{ inset } m2 \text{ | } x \text{ in set } m1\end
  \]

  It is declared infix, of precedence 4.

- **submap** : \(\forall 'a. 'a 
  \rightarrow \text{bool}\) tests the inclusion of maps: it returns true if the first map is a submap of the second. It may be defined as:

  \[
  \text{fun op submap } (m1,m2) = 
  \text{ all } x \text{ inset } m2 \text{ andalso } ?m1(x) = ?m2(x) \text{ | } x \text{ in set } m1\end
  \]

  It is declared infix, of precedence 4.

- **dom** : \(\forall 'a. 'a 
  \rightarrow 'a\) set computes the domain of a map; it is the identity on sets. It is syntactic sugar for:

  \[
  \text{fun dom } m = \{x \mid x \text{ in set } m\}\end
  \]

- **rng** : \(\forall 'a. 'a 
  \rightarrow 'b\) set computes the range of a map. The range of a non-empty set is always \(\{()\}\). **rng** is syntactic sugar for:
fun rng m = {y | _ => y in map m}

- **card : (''a -m> 'b) -> int** computes the cardinal of a map. It may be defined as:

  fun card {} = 0 | card {_ => _} U m' = 1+card m'

- **<| : (''a -m> 'c) * (''a -m> 'b) -> (''a -m> 'b)** is the “domain restrict to” function. If s is a map and m is another map, then s <| m is the map m', whose domain is restricted to the elements in the domain of s. This may be defined by:

  fun op <| (s,m) = {x => y | x => y in map m such that x inset s}

  It is declared infix, right associative of precedence 7.

- **<|- : (''a -m> 'c) * (''a -m> 'b) -> (''a -m> 'b)** is the “domain restrict by” function. If s is a map and m is another map, then s <-| m is the map m, whose domain is restricted to the elements outside the domain of s. This may be defined by:

  fun op <-| (s,m) = {x => y | x => y in map m such that not (x inset s)}

  It is declared infix, right associative of precedence 7.

- **|> : (''a -m> 'b) * (''b -m> 'c) -> (''a -m> 'b)** is the “range restrict to” function. If s is a map and m is another map, the m |> s is the map m', where only the maplets x => y such that y is in the domain of s are considered. This may be defined by:

  fun op |> (m,s) = {x => y | x => y in map m such that y inset s}

  It is declared infix, left associative of precedence 7.

- **|-> : (''a -m> 'b) * (''b -m> 'c) -> (''a -m> 'b)** is the “range restrict by” function. If s is a map and m is another map, the m |-> s is the map m, where only the maplets x => y such that y is outside the domain of s are considered. This may be defined by:

  fun op |-> (m,s) = {x => y | x => y in map m such that not (y inset s)}

  It is declared infix, left associative of precedence 7.

- **++ : (''a -m> 'b) * (''a -m> 'b) -> (''a -m> 'b)** is the map overwriting operator. It takes two maps m and m', and returns a map m'', whose domain is the union of the domains of m and m', and which maps every element x of the domain of m' to ?m' (x), and every other element x to ?m (x). Thus m'' is m, over which the associations of m' have been written. To underwrite instead of overwriting, write m' ++ m instead of m ++ m'. The only difference is that m will be evaluated before m.

  ++ is declared infix, left associative of precedence 6.

- **overwrite : (''a -m> 'b) list -> ''a -m> 'b** is the distributed overwriting function. It returns the first map in the list, overwritten by the second, the third, etc. It is equivalent to:

  fun overwrite nil = ()
  | overwrite (ml::rest) = ml ++ overwrite rest
The application of overwrite to a list comprehension is compiled in an optimized form, where the overwritings are done on the fly, without building the list comprehension first.

- **overwrite** : (''a -m> 'b) list -> ''a -m> 'b is the distributed writing function. It returns the last map in the list, overwritten by the next to last, the penultimate, etc. It is equivalent to:

  fun overwrite nil = {}
  | overwrite (ml::rest) = overwrite rest ++ ml

The application of overwrite to a list comprehension is compiled in an optimized form, where the underwritings are done on the fly, without building the list comprehension first.

Note that overwrite o rev=underwrite, and underwrite o rev=overwrite.

- **underwrite** : (''a -m> 'b) list -> ''a -m> 'b is the distributed underwriting function. It returns the last map in the list, overwritten by the next to last, the penultimate, etc. It is equivalent to:

  fun underwrite nil = {}
  | underwrite (m1::rest) = underwrite rest ++ m1

The application of underwrite to a list comprehension is compiled in an optimized form, where the underwritings are done on the fly, without building the list comprehension first.

Note that overwrite o rev=underwrite, and underwrite o rev=overwrite.

- **delta** : (''a -m> 'b) * (''a -m> 'b) -> (''a -m> 'b) computes the symmetric difference of two maps. This is the overwriting of one map by another, restricted by the intersection of the domains. It may be defined by:

  fun op delta (m,m') = (m' <-| m) ++ (m <-| m')

(or equivalently, (m <-| m') ++ (m' <-| m)). It generalizes the classical notion of symmetric difference of sets. It is declared infix, left associative of precedence 7.

- **choose** : (''a -m> 'b) -> ''a chooses an element in the domain of the map in argument. It raises Empty if the map is empty. It is syntactic sugar for:

  fun choose {} => raise Empty
  | choose {x => _,...} = x

or for:

  fun choose m = case (some x | x in set m end) of
    SOME x => x
  | NONE => raise Empty

- **choose_rng** : (''a -m> 'b) -> 'b chooses an element in the range, and raises Empty if the map is empty. The element chosen in the range is precisely the image by the map of the one chosen by choose, as shows the equivalent form:

  fun choose_rng m = ?m(choose m)

- **split** : (''a -m> 'b) -> (''a -m> 'b) * (''a -m> 'b) splits a map in two disjoint maps, whose union is the original one. In general, the splitting won’t yield maps of equal (or nearly equal) cardinalities. However, the splitting has the following properties:

  - Splitting a map of cardinality greater than or equal to 2 yields two non empty maps. Hence, recursive procedures that decompose their map arguments with split until its cardinality goes down to 1 or 0 will always terminate.
  - If (m1,m2)=split m, then all elements in the domain of m1 are less than elements in the domain of m2 in the system order.
  - Splitting depends only on the domain, not on the range. That is, assume if dom m=dom m', and both (m1,m2)=split m and (m'1,m'2)=split m', then dom m1=dom m'1 and dom m2=dom m'2.

  29
Splitting has the statistical property that if two maps have similar domains (that is, their domains differ for a small number of elements), recursively splitting both domains will on average unearth an optimum number of common subdomains. For a more detailed description of this property, refer to [11].

As a consequence, recursive memo functions with map arguments that recurse through splittings of their arguments should be good incremental functions, recomputing quickly their results on data similar to previous similar data.

- \( \mathbf{U} : 'a \text{ set} \times 'a \text{ set} \rightarrow 'a \text{ set} \)
  is the union of two sets. It may be defined as a special case of map overwriting, because sets are special cases of maps:

  \[
  \text{fun op \( \mathbf{U} \) (s : 'a \text{ set}, s' : 'a \text{ set}) = s ++ s'}
  \]

  (or equivalently, \( s' ++ s \)). It is declared infix, left associative, of precedence 6.

- \( \mathbf{\&} : 'a \text{ set} \times 'a \text{ set} \rightarrow 'a \text{ set} \)
  is the intersection of two sets. It may be defined as a special case of map domain restriction, because sets are special cases of maps:

  \[
  \text{fun op \( \mathbf{\&} \) (s : 'a \text{ set}, s' : 'a \text{ set}) = s <\mid s'}
  \]

  It is declared infix, left associative, of precedence 7.

- \( \text{intersects} : ('a \rightarrow 'b) \times ('a \rightarrow 'c) \rightarrow \text{bool} \)
  returns true if its two arguments have domains with a non-empty intersection. It can be defined as:

  \[
  \text{fun op \( \text{intersects} \) (m, m') = not (empty (m <\mid m'))}
  \]

  It is declared infix, of precedence 4.

- \( \text{\textbackslash} : 'a \text{ set} \times 'a \text{ set} \rightarrow 'a \text{ set} \)
  is the difference of two sets. It may be defined as a special case of the domain restrict by operator, because sets are special cases of maps:

  \[
  \text{fun op \( \text{\textbackslash} \) (s : 'a \text{ set}, s' : 'a \text{ set}) = s' <\mid s}
  \]

  It is declared infix, left associative, of precedence 7.

- \( \text{union} : ('a \rightarrow 'b) \rightarrow 'a \text{ set} \)
  is the distributed union of the sets in the domain of the argument, it is quite similar to the overwrite and underwrite functions. It can be defined as:

  \[
  \text{fun \( \text{union} \) {} = {}}
  | \text{union \{s => _\} = s}
  | \text{union ss = let val (s1, s2) = split ss}
  | \text{in union s1 U union s2}
  \text{end}
  \]

  The application of \( \text{union} \) to a set or map comprehension is compiled in an optimized form, where the unions are done on the fly, without building the set comprehension first.

- \( \text{inter} : ('a \rightarrow 'b) -> ('a \rightarrow 'c) \rightarrow ('a \rightarrow 'b) \)
  is the distributed intersection of maps (and hence of sets, too) in the domain of its arguments. It is mostly used when \( 'b \) and \( 'c \) are both \texttt{unit}, in which case it computes the distributed intersection of a set of sets. It raises \texttt{Empty} on the empty set. It can be defined as:
fun inter {} = raise Empty
   | inter {s => _} = s
   | inter ss = let val (s1,s2) = split ss
                 in
                   inter s1 & inter s2
                 end

The application of \texttt{inter} to a set or map comprehension is compiled in an optimized form, where the intersections are done on the fly, without building the set comprehension first.

- \texttt{mapadd} : (''a * 'b) * (''a -m> 'b) -> (''a -m> 'b) adds one maplet to a map, overwriting the map. It may be defined as:

\[
\text{fun mapadd} \ (\ (x,y),m) = m ++ \ (x => y)\]

It is only a bit faster than writing \(m ++ (x => y)\).

- \texttt{mapaddunder} : (''a * 'b) * (''a -m> 'b) -> (''a -m> 'b) adds one maplet to a map, underwriting the map. It may be defined as:

\[
\text{fun mapaddunder} \ (\ (x,y),m) = \{x => y\} ++ m\]

It is only a bit faster than writing \(\{x => y\} ++ m\) (and the order of evaluation is different).

- \texttt{mapremove} : ''a * (''a -m> 'b) -> (''a -m> 'b) removes a maplet from a map. It may be defined as:

\[
\text{fun mapremove} \ (x,m) = \{x\} <-| m\]

It is only a bit faster than writing \(\{x\} <-| m\).

- \texttt{inv} : (''a -m> ''b) -> (''b -m> ''a) inverses a map. Its definition is the same as:

\[
\text{fun inv} \ m = \{y => x \mid x => y \text{ in map } m\}\]

so in case \(m\) is not invertible, \texttt{inv} returns a map that maps \(y\) to the largest \(x\) in the system order such that \(m\) maps \(x\) to \(y\).

- \texttt{O} : (''b -m> 'c) * (''a -m> ''b) -> (''a -m> 'c) is the composition of maps. It is precisely defined by:

\[
\text{fun op O} \ (m,m') = \{x => ?m(y) \mid x => y \text{ in map } m' \text{ such that } y \text{ inset } m\}\]

It is declared infix, left associative of precedence 3. It is the map version of the \texttt{o} function composition operator.

- \texttt{to} : int * int -> int set is the range function. If \(a\) and \(b\) are the first and second argument respectively, \texttt{to} returns the set of all integers \(x\) such that \(a \leq x \leq b\). We could have defined \texttt{to} by:

\[
\text{fun op to} \ (a,b) = \begin{cases} 
\{\} & \text{if } x>b \\
\{x\} U f(x+1) & \text{else } \{x\} \end{cases}
\]

in
   f a
end
to is declared infix, left associative of precedence 9, so that we may write a to b.

• mapoflist : ‘a list -> (int -> ‘a) converts a list to a map from its indices to its elements. It may be defined by:

```haskell
fun mapoflist l = 
  let fun f(_,nil) = {}
       | f(n,a::l) = {n => a} ++ f(n+1,l)
  in
  f(0,l)
end
```

• inds : ‘a list -> int set computes the set of indices of a list l, i.e, \{0, ..., len l-1\}. It may be defined by:

```haskell
fun inds l = 0 to (len l-1)
or by val inds = dom o mapoflist.
```

• elems : ‘a list -> ‘a set computes the set of elements of the list in argument. It may be defined by:

```haskell
fun elems l = {x | x in list l}
or by val elems = rng o mapoflist.
```

There is now an alternative data type for maps in HimML, the table type. This implements imperative maps: unlike the map type, objects of type table are updated in a destructive way, just like hash-tables. Objects of type table, which we shall just call tables, are not really faster than using applicative references, but they put less stress on the sharing mechanism and the garbage collector of HimML, which may result in space and even time savings.

The type of tables mapping objects of type ‘‘a to ‘b is

```haskell
type (''a, 'b) table
```

You can think as a kind of equivalent of the type (‘‘a -> 'b) ref. Associated functions are:

• table : unit -> (''_a, '_b) table creates a fresh, empty table. If tables were implemented as objects of type (''a -> 'b) ref, this would be equivalent to:

```haskell
fun table () = ref {};
```

• t_get : (''a, 'b) table -> 'b -> 'a option reads an element off a table. Precisely, t_get t x returns SOME y if x is mapped to y by the table t, and NONE if x has no entry in t. If tables were implemented as objects of type (''a -> 'b) ref, this would be equivalent to:

```haskell
fun t_get t x = 
  SOME (?!t x) handle MapGet => NONE;
```

• t_put : (''a, 'b) table -> 'a * 'b -> unit, called as t_put t (x, y) adds a new binding from x to y to the table t, erasing any previously existing binding for x. If tables were implemented as objects of type (''a -> 'b) ref, this would be equivalent to:

```haskell
fun t_put t (x, y) = 
  t := !t ++ {x => y};
```
• \texttt{t\_put\_behind} : \((\text{'a}, \text{'b})\) table \to \text{'a} \times \text{'b} \to \text{unit}, called as \text{t\_put\_behind} \ t \ (x, y)\) adds a new binding from \(x\) to \(y\) to the table \(t\), except if any binding for \(x\) already existed. If tables were implemented as objects of type \((\text{'a} - \text{m} \to \text{'b})\) ref, this would be equivalent to:

\[
\text{fun t\_put\_behind t (x, y) =} \\
\quad t := (x \Rightarrow y) ++ !t;
\]

• \texttt{t\_remove} : \((\text{'a}, \text{'b})\) table \to \text{'a} \to \text{unit}, removes any entry associated with its second argument from the table given in first argument. If tables were implemented as objects of type \((\text{'a} - \text{m} \to \text{'b})\) ref, this would be equivalent to:

\[
\text{fun t\_remove t x =} \\
\quad t := (x) <-| !t;
\]

• \texttt{t\_iter} : \((\text{'a}, \text{'b})\) table \to \((\text{'a} \times \text{'b} \to \text{bool}) \to \text{bool}\) is the standard iterator over tables. This is meant to implement a form of existential quantification, but can be used to loop over the table, and doing some computation on each entry, just like the \text{iterate} quantifier. Calling \text{t\_iter} \ t \ f\) iterates over all elements of the table \(t\), in some indefinite order, calls \(f\) \((x,y)\) for each entry \(x \Rightarrow y\) in \(t\). This stops when the table has been fully traversed, and then returns \text{false}, or after the first call to \(f\) returns \text{true}, in which case \text{iter} \ t \ f\) returns \text{true}.

If tables were implemented as objects of type \((\text{'a} - \text{m} \to \text{'b})\) ref, this would be equivalent to:

\[
\text{fun t\_iter t f =} \\
\quad \text{exists} \\
\quad f \ (x, y) \\
\quad | x \Rightarrow y \in \text{map} \ !t \\
\quad \text{end;}
\]

This can be used to implement \text{iterate}-style loops, by writing

\[
\text{t\_iter t (fn (x, y) => (f(x,y); false)); ()}
\]

Warning: it is definitely a bad idea to modify the table \(t\) while you iterate on it.

• \texttt{t\_collect} : \((\text{'a}, \text{'b})\) table \to \((\text{'a} \times \text{'b} \to (\text{'c} - \text{m} \to \text{'d})) \to (\text{'c} - \text{m} \to \text{'d})\) is an iterator over tables, just like \text{t\_iter}. This one is rather meant to implement set and map comprehensions: \text{t\_collect} \ t \ f\) iterates over all elements of the table \(t\), in some indefinite order (although it is guaranteed to be the same as the one used by \text{t\_iter}), calls \(f\) \((x,y)\) for each entry \(x \Rightarrow y\) in \(t\), and computes the \text{overwrite} of all maps returned by each call to \(f\).

If tables were implemented as objects of type \((\text{'a} - \text{m} \to \text{'b})\) ref, this would be equivalent to:

\[
\text{fun t\_collect t f =} \\
\quad \text{overwrite} \ [f \ (x, y) \\
\quad \mid x \Rightarrow y \in \text{map} \ !t];
\]

For example, getting the contents of the table \(t\) as a map can be effected as:

\[
\text{t\_collect t (fn (x, y) => (x => y))}
\]

Building the map of all \(x \Rightarrow y\) in \(t\) such that the predicate \(P\) \((x,y)\) holds can be done by:

\[
\text{t\_collect t (fn (x, y) => if P (x,y) then (x => y) else {})}
\]

33
Warning: it is definitely a bad idea to modify the table \( t \) while you iterate on it.

- \( \text{t\_reset} : (\langle \alpha, \beta \rangle \text{ table} \to \text{unit} \) resets the table in argument. If tables were implemented as objects of type \( (\langle \alpha \rightarrow \beta \rangle \text{ ref} \) this would be equivalent to:

\[
\text{fun t\_reset t = (t := \{\});}
\]

4.4 Refs and Arrays

Refs and arrays are the fundamental mutable data structures of HimML, as in Standard ML of New Jersey (only refs exist in the definition of Standard ML). Ref and array types always admit equality, even if their argument types do not. Equality is decided according to the following rules:

- Every ref or array is equal to itself.
- Any newly created ref (by \( \text{ref} \)), or newly created array (by \( \text{array or arrayoflist or iarray or iarrayoflist} \)), is different from any other ref or array.

In short, refs and arrays are not shared, and are compared by their addresses\(^1\). Some syntax extensions are defined to allow you to write a more readable code to handle arrays. In particular:

- \( [\langle x_1, \ldots, x_n \rangle] \) denotes the array containing \( x_1, \ldots, x_n \) in this order. This is an abbreviation for \( \text{arrayoflist} [x_1, \ldots, x_n] \)

Furthermore, the compiler actually does not build the list to convert it to an array afterwards, but builds the array directly. This is also the syntax used by the pretty-printer to print out arrays.

- \( a.\ (n) \) abbreviates \( \text{sub} \ (a, n) \), and denotes the element at position \( n \) (starting from 0) in array \( a \).
- \( a.\ (n) \ .:= \ e \) abbreviates \( \text{update} \ (a, n, e) \), i.e. it stores the value of \( e \) in array \( a \) at position \( n \). Note that \( a \) need not be a variable name, but can be any object of type an array.

There is however no similar syntactic sugar for non-polymorphic integer arrays (of type \( \text{intarray} \)), only for polymorphic arrays (of type \( \langle \alpha \rangle \text{ array} \), for any \( \langle \alpha \rangle \)).

The following exception is defined:

\text{exception Subscript}

It is raised when accessing an element outside of an array, by \( \text{sub or update} \), or by \( \text{isub or iupdate} \). The following functions are provided:

- \( \text{ref} : \langle \alpha \rangle \to \langle \alpha \rangle \text{ ref} \) creates a new mutable reference.
- \( \ := \ : \langle \alpha \rangle \text{ ref} * \langle \alpha \rangle \to \text{unit} \) modifies the value of a reference, that is, assigns the value of the second argument to the reference in first argument. It is declared infix, of precedence 3.
- \( ! : \langle \alpha \rangle \text{ ref} \to \langle \alpha \rangle \) is the dereferencing function. It gets the value stored inside a reference. It could have been defined as:

\[
\text{fun ! (ref v) = v}
\]

because \( \text{ref} \) is understood in ML as a (fake) datatype constructor.

\(^1\)Actually, all objects in HimML are compared by their addresses, because any two equal objects are located at the same address.
• array : int * '1a -> '1a array creates an array with n elements equal to e, where n and e are the arguments. If n is negative, the exception Subscript is raised. n is called the length of the array.

• sub : 'a array * int -> 'a dereferences the n-th element of the array a (indices start at 0), where a and n are the arguments. If n is negative or greater than or equal to the length of the array, the exception Subscript is raised.
  A more readable notation for sub (a, n) is a.(n).

• update : 'a array * int * 'a -> unit modifies the n-th element of the array a, replacing it by the value e, where a, n and e are the arguments. If n is negative or greater than or equal to the length of the array, the exception Subscript is raised.
  A more readable notation for update (a, n, e) is a.(b) := e.

• length : 'a array -> int gets the length of the array in argument.

• arrayoflist : '1a list -> '1a array converts a list into an array with the same elements in the same order. The length of the resulting array is exactly the length of the argument list.

• iarray : int * int -> intarray creates an array with n integer elements equal to e, where n and e are the arguments. If n is negative, the exception Subscript is raised. n is called the length of the array.

• isub : intarray * int -> int dereferences the n-th element of the integer array a (indices start at 0), where a and n are the arguments. If n is negative or greater than or equal to the length of the array, the exception Subscript is raised.

• iupdate : intarray * int * int -> unit modifies the n-th element of the array a, replacing it by the integer value e, where a, n and e are the arguments. If n is negative or greater than or equal to the length of the array, the exception Subscript is raised.

• ilength : intarray -> int gets the length of the array of integers in argument.

• iarrayoflist : int list -> intarray converts a list into an array with the same integer elements in the same order. The length of the resulting array is exactly the length of the argument list.

4.5 Strings

Strings are ordered sequences of characters (we call size of the string the length of the sequence). ML does not provide a type of characters, though one-character strings may serve this purpose. Strings are assumed coded according to the 7-bit ASCII standard. However, characters are usually 8 bits wide: the characters of code greater than 127 are interpreted in a system-dependent fashion. The string length is encoded separately from the sequence of characters that composes it: no special character is reserved to represent the end of a string, in particular, any string may contain the NUL character (\000).

The following exceptions are defined:

• exception Ascii, raised when using a number that is not the code of an ASCII code (with 8-bit characters, a number that is not an integer between 0 and 255).

• exception StringNth, raised when accessing an element outside a string.

• exception RE of int, raised with an integer error code when the regexp function encounters an error.
  The remsg function can be used to get a plain text description of the error.

The functions are:
• `explode : string -> string list` converts a string into the list of its characters, where characters are represented as one element strings. For instance `explode "abc"` is `"a", "b", "c"`.

• `implode : string list -> string` is the inverse of `explode`. If given a list of characters (one-character strings), it produces the corresponding string. Actually, it accepts any list of strings, whatever their size, and concatenates them, as a generalization of the intended semantics.

• `^ : string * string -> string` concatenates two strings. It could have been defined as:

```plaintext
fun op ^ (s, s') = implode (explode s @ explode s')
```

(or `implode [s, s']`, noticing the generalization of `implode` though `^` is more efficient. `^` is declared infix, right associative of precedence 6.

• `concat : string list -> string` is the distributed concatenation of strings. Because of the general character of `implode`, `implode` and `concat` are synonymous.

• `size : string -> int` computes the size of a string. This may be defined by:

```plaintext
fun size s = len (explode s)
```

although it does not build the exploded list.

• `chr : int -> string` builds the string having as only character the character with code `n` given as argument. If `n` is not the code of a character, the exception `Ascii` is raised.

• `ord : string -> int` gets the code of the first character of the string. The exception `StringNth` is raised if the string is empty (""").

• `ordof : string * int -> int` gets the code of the `n+1`th character in the string `s`, where `s` and `n` are the arguments. The index `n` starts from 0. If `n < 0` or `n` is greater than or equal to the size of `s`, the exception `StringNth` is raised. This is equivalent to:

```plaintext
fun ordof (s, n) = ord (explode s nth n) handle Nth => raise StringNth
```

• `substr : string * int * int -> string` extracts a substring from the string `s`, given as first argument. Call `i` the second argument, and `j` the third. `substr(s, i, j)` returns the string of all characters from indices `i` included to `j` excluded. `substr` raises `StringNth` if `i < 0` or `j > len s`, or `i` or `j` is not integer. It may be defined as:

```plaintext
fun substr (s, i, j) = 
  let fun f k = 
    if k>=j
      then ""
    else chr (ordof (s, k)) ^ f(k+1)
  in
  f i
end
```

• `strless : string * string -> bool` compares strings in lexicographical order, the order on characters being defined by the order on their codes. Equivalently:
fun op strless (s,s') = 
  let fun less (_,nil) = false 
      | less (nil,_) = true 
      | less (c::l,c'::l') = 
        ord c<ord c' orelse 
        ord c=ord c' andalso less(l,l') 
  in 
    less (explode s,explode s') 
  end

strless is declared infix, of precedence 4.

• regexp : string -> 
  |[match : string -> (int * int) option, 
     matches : string -> bool, 
     matchsub : string * int * int -> (int * int) option, 
     subst : string -> string]| builds a regular expression matcher and substitution engine, on the model of the Unix grep utility. This is based on Harry Spencer’s regexp(3) package.

regexp re builds a non-deterministic finite automaton with some optimizations for recognizing the language defined by the regular expression re. It returns a record with four methods, match, matches and matchsub for matching, and subst for substituting substrings matched previously by match in a given string.

The match function, applied on a string s to be searched, looks for the first (leftmost) occurrence of a substring of s that is matched by the regular expression re. It returns SOME (i,j) if it has found one, and then substring (s,i,j) is the matched substring. If it cannot find one, it returns NONE.

For example:

    val [match,...] = regexp "yp";

builds a match function for finding the first occurrence of "yp" in a string. Then, to look for an occurrence of the latter in various strings:

    match "type checking";

returns SOME (1,3).

The syntax of regular expressions is mostly as in any other regexp package. The special characters are:

- \^ matches the empty string, but only at the beginning of the string s to be matched.
- \$ similarly, matches the empty string, but at the end of the string s to be matched. Therefore, to test whether a string s is in the language defined by re, it is enough to test:

    case #match (regexp "\^\$") (s,0,size s) of SOME _ => true | _ => false

- \< matches the empty string at the beginning of a word (any sequence of letters, digits and underscores).
- \> matches the empty string at the end of a word.
- \b matches the empty string at the beginning or end of a word.
- \B matches the empty string everywhere except at the beginning or end of a word.
- \w matches any word-constituent character (a letter, a digit, or an underscore).
- \W matches any non word-constituent character (anything but a letter, a digit, or an underscore).
- `\` (one backslash character) can be used to quote the next character. This is useful when you wish to include a special character, such as `.`, as an ordinary character.
- `.` matches any single character.
- `[..]` defines a character group, so that `[a-z]` matches any letter in small caps, `[^A-Z]` matches any character that is not a capital letter, and so on.
- `(.....)` defines a group, on which operators like `*` or `+` for example can be applied as a whole. Groups can also be used to mark matched substrings for future reference by the `subst` function. For example:
  ```
  val `[match,subst,...]` = regexp "^foo)*+(a+)(bar)*$";
  match "foofooaoaadbaar";
  subst "\0;\1;\2;\3";
  returns "foofooaoaadbaar;foo;aaaaaa;bar".
  ```
- `\1, ..., \9` can also be used to match whatever was last recognized by the corresponding parenthesized group. For example:
  ```
  val `[match,subst,...]` = regexp "^a*)b\1$";
  match "aaabaaa";
  subst SOME (0,7), and indeed "aaabaaa" is a group of `n` as (here, `n = 3`), followed by a `b`, and then the same group of `n` as. This can be used to recognize sets of words that are non regular. (This feature does not work in Harry Spencer’s version of the regexp package on which the HimML version is based, but works in HimML.)
- `*` is a suffix. If `g` is a letter, a character group or a parenthesized group, `g*` matches any sequence of zero or more words, each one matched by `g`.
- `+` is as `*`, except that it matches a sequence of at least one word.
- `?` is as `*`, except that it matches a sequence of at most one word.

The `matches` function is a trimmed down version of `match`, which only returns whether there is a match or none. So `#matches (regexp re) s` is equivalent to:
```
(case #match (regexp re) s of SOME _ => true | NONE => false)
```

The `matchsub` function is, on the contrary, a puffed up version of `match`, which does not examine the whole string to be matched, but only a substring: `#matchsub (regexp re) (s,i,j)` looks for a substring of `s` that would be matched by `re`, but only between positions `i` and `j`. It then returns `SOME (i',j')` if it has found a match: `i'` and `j'` are the bounds for the leftmost match between positions `i` and `j`. Otherwise, it returns `NONE`. Notice that `i'` and `j'` are positions in `s`, not in the substring `substr (s,i,j)`. In fact, `#matchsub (regexp re) (s,i,j)` is equivalent to:
```
(case #match (regexp re) (substr (s,i,j)) of
  SOME (i',j') => SOME (i+i',i+j')
  | NONE => NONE)
```

As `substr, matchsub` raise the exception `StringNth` if the positions `i, j` are out of bounds.

Any of these functions may raise the exception `RE n`, where `n` is an error code. `regexp` itself may raise it, when the regular expression `re` contains a syntax error, or contains too deeply nested parentheses (the current limit is 10 levels), etc. The `remsg` function provides a hopefully legible error message.

A possible bug is that NUL characters (of code 0) may be recognized erratically. It is better not to include any NUL characters in either the regular expression or the string to be matched.
• `remsg : int -> string` translates the regular expression error code in argument (as returned as argument of a `RE` exception) into a string in human-readable form.

• `intofstring : string -> int` converts the input string into an integer. This assumes the integer is written in decimal; minus signs may be written in standard notation (−) or as in ML (˜). No error is returned, even if the input string is not the decimal representation of an integer. To check this, call `regexp "[−]?[0−9]+"` first.

Note: to convert back an integer `i` to a string, use

```ml
let val f as "convert, ..." = outstring ""
  in
    print f (pack (i:int));
    convert ()
  end
```

• `numofstring : string -> num` converts the input string into a real number. This assumes the real is written in decimal; minus signs may be written in standard notation (−) or as in ML (˜). No error is returned, even if the input string is not the decimal representation of a real. To check this, call

```ml
regexp "([-]?[0−9]+(\.?[0−9]*)?(e[EFfgG][−]?[0−9]+)?)" first.
```

Note: to convert back a number `x` to a string, use

```ml
let val f as "convert, ..." = outstring ""
  in
    print f (pack (x:num));
    convert ()
  end
```

### 4.6 Integer Arithmetic

HimML integers are machine integers, that is, integers in the range `[min_int, max_int]`, and are represented with `nbits` bits.

Exceptions on integer operations can be:

• `exception Arith`, which is raised whenever an arithmetic error occurs, for example division by 0. (Overflow in addition or multiplication is normally not dealt with, and just produces wrong results—more precisely, results modulo the word size—on most machines. This is implementation-dependent.)

Non-negative integers are 0, 1, 2, ..., 42, ... Negative integers are written with the ˜ negation operator in front.

The following operations are defined. Basic operations like +, −, etc. are not overloaded as in Standard ML. For example, + is always an integer function. For analogous numerical (floating-point) functions, see Section 4.7. For example, floating-point addition is `#+`, not `+`.

• `+ : int * int -> int` is integer addition. It is declared infix, left associative of precedence 6.

• `− : int * int -> int` is integer subtraction. It is declared infix, left associative of precedence 6.

• `∗ : int * int -> int` is integer multiplication. It is declared infix, left associative of precedence 7.

• `˜ : int -> int` is integer negation.
• \texttt{sqr} : \texttt{int} \rightarrow \texttt{int} squares its argument.
• \texttt{abs} : \texttt{int} \rightarrow \texttt{int} computes the absolute value of its argument.
• \texttt{<} : \texttt{int} \times \texttt{int} \rightarrow \texttt{bool} returns true if its first argument is less than its second argument.
• \texttt{>} : \texttt{int} \times \texttt{int} \rightarrow \texttt{bool} returns true if its first argument is greater than its second argument.
• \texttt{<=} : \texttt{int} \times \texttt{int} \rightarrow \texttt{bool} returns true if its first argument is less than or equal to its second argument.
• \texttt{>=} : \texttt{int} \times \texttt{int} \rightarrow \texttt{bool} returns true if its first argument is greater than or equal to its second argument.
• \texttt{min} : '^	exttt{int}' * '^	exttt{int}' -> '^	exttt{int}' returns the least of its two arguments.
• \texttt{max} : \texttt{int} \times \texttt{int} \rightarrow \texttt{int} returns the greatest of its two arguments.
• \texttt{div} : \texttt{int} \times \texttt{int} \rightarrow \texttt{int} is quotient extraction. It is declared infix, left associative of precedence 7. The definition conforms to the definition of Standard ML, where the quotient has the smallest possible absolute value (the remainder has the same sign as \texttt{x}).
• \texttt{mod} : \texttt{int} \times \texttt{int} \rightarrow \texttt{int} is the remainder operation.
  It is syntactic sugar for \texttt{fn} (\texttt{x},\texttt{y}) => \texttt{x} - \texttt{y}*(\texttt{x div y}). It is declared infix, left associative of precedence 7.
• \texttt{divmod} : \texttt{int} \times \texttt{int} \rightarrow \texttt{int} \times \texttt{int} returns both the quotient and remainder of the integers in argument. This could be defined as \texttt{fn} (\texttt{x},\texttt{y}) => (\texttt{x div y},\texttt{x mod y}), but is normally faster.
• \texttt{bor} : \texttt{int} \times \texttt{int} \rightarrow \texttt{int} computes the binary inclusive or (the disjunction) of the two integers in argument. There is a one bit at some position in the result if there was a one at the same position in at least one of the argument integers.
• \texttt{bxor} : \texttt{int} \times \texttt{int} \rightarrow \texttt{int} computes the binary exclusive or of the two integers in argument. There is a one bit at some position in the result if there was a one at the same position in exactly one of the argument integers.
• \texttt{band} : \texttt{int} \times \texttt{int} \rightarrow \texttt{int} computes the binary and (the conjunction) of the two integers in argument. There is a one bit at some position in the result if there was a one at the same position in both of the argument integers.
• \texttt{bnot} : \texttt{int} \rightarrow \texttt{int} computes the binary negation of the integer in argument. There is a one but at some position in the result if there was a zero at the same position in the argument.
• \texttt{lsl} : \texttt{int} \times \texttt{int} \rightarrow \texttt{int} computes the logical shift left of the first integer by the number of bits specified in the second argument. Its behavior is unspecified if the number of bits is negative.
• \texttt{lsr} : \texttt{int} \times \texttt{int} \rightarrow \texttt{int} computes the logical shift right of the first integer by the number of bits specified in the second argument. Its behavior is unspecified if the number of bits is negative.
• \texttt{asr} : \texttt{int} \times \texttt{int} \rightarrow \texttt{int} computes the arithmetic shift right of the first integer by the number of bits specified in the second argument (arithmetic means that the sign bit is kept as is, and propagates through the shifting). Its behavior is unspecified if the number of bits is negative.
• \texttt{inc} : \texttt{int ref} \rightarrow \texttt{unit} increments the integer pointed to by the reference in argument.
• \texttt{dec} : \texttt{int ref} \rightarrow \texttt{unit} decrements the integer pointed to by the reference in argument.
• **rand** : unit -> int draws a pseudo-random equidistributed number in the integer interval \([\text{min}_\text{int}, \text{max}_\text{int}]\). The method used is based on a generalized feedback shift register [8], with generating polynomial \(x^{55} + x^{24} + 1\). The period is \(2^{55} - 1\), which means that, practically, the generator never loops back to any previous state.

• **irand** : int -> int draws a pseudo-random equidistributed number in the integer interval \([0, x]\), where \(x\) is the given argument. If \(x \leq 0\), it instead draws an equidistributed pseudo-random integer in \([0, \text{max}_\text{int}] \cup [\text{min}_\text{int}, x]\) (think of integers as two’s complement integers). So if \(x = 0\), this is equivalent to **rand**. This calls **zrand** anyway.

This can be used to draw equidistributed integers in \([a, b]\) (with \(a < b\)) by calling \(a + \text{irand}(b - a)\).

### 4.7 Numerical Operations

The real and imaginary parts of numbers are floating-point numbers, which are classified according to the IEEE754 standard. A floating-point number may be:

• zero (0.0). In HimML, the negation of 0.0 is exactly 0.0, that is 0.0 and \(-0.0\) are identified in HimML (this is not so in the IEEE754 specification).

• a normalized, positive or negative number. These are ordinary numbers (significant, non-zero, not too small, not too large).

• a denormalized number, that is a number whose absolute value is too small to be represented as a normalized number, but not small enough to be mistaken for zero. Denormalized numbers will only appear in IEEE754 implementations of HimML.

Denormalized numbers are useful because they make the rounding of small numbers degrade smoothly as these tend towards zero. Numerically speaking, testing a number against being zero or denormalized is the right way to decide if we can divide by it.

• infinity, whether it be \(+\text{inf}\) or \(-\text{inf}\). There are no infinities in non-IEEE754 systems. Dividing by zero a non-zero number \(x\) yields an infinity, positive if \(x > 0\), negative if \(x < 0\) in an IEEE754 system, and raises the **Arith** exception in non-IEEE754 systems.

• a NaN (Not a Number). NaNs represent numerical errors (dividing 0 by 0, taking the square root of a negative real number, etc.), and are provided only in IEEE754 systems; on other systems, an **Arith** exception is raised instead.

The fact that an infinity or a NaN is produced (in IEEE754 systems), or an **Arith** exception is raised (in non-IEEE754 systems) by a computation means usually that the computation is wrong. In this case, the exception system is better suited to find numerical bugs, provided we have a means of locating the place where the exception is raised.

However, in production code, these error conditions may still crop up for limit situations. It is then unacceptable for code to stop functioning because of this: so infinities and NaNs should be used. In general, users had better use a system obeying the IEEE754 standard (or its successors). The IEEE754 standard defines trapping mechanisms to be used for debugging numerical codes; they are not used in HimML now, but may be in a future version of the debugger.

Classification of numbers is accomplished with the auxiliary types:

```haskell
datatype numClass = MNaN of int | MInf | MNormal | MDenormal | Zero | PDenormal | PNormal | PInf | PNaN of int
```

**numClass** encodes both the class and the sign of a real number. In the case of a NaN, the NaN code is also provided as an integer. NaN codes are system-dependent; only NaN codes from 1 to 7 are recognized in HimML; unrecognized codes are represented by 0. NaN codes and the constructors **MNaN**, **MInf**, **MDenormal**, **PDenormal**, **PInf**, **PNaN** are never used in a non-IEEE754 system. The exception:
exception Arith

is raised for all arithmetical errors in non-IEEE754 systems, and is never used in IEEE754 systems.

On a different subject, remember that testing complex numbers for equality is hazardous. Therefore, \( = \) is not to be used lightly on numbers (for example, \( 0.2 \times 5 \) may print as \( 1 \) but differ from \( 1 \) internally). Normally, though, computations on sufficiently small integers should be exact (at least on real IEEE754 systems). This warning applies not only to the explicit \( = \) equality function, but also to all operations that use internally the equality test, in particular the test for being inside a set \( \text{inset} \) or the application of a map to an object \( ? \), when this object contains numbers.

Moreover, following the principle, valid in HimML, that an object is always equal to itself, \( +\text{inf} = +\text{inf} \), \( -\text{inf} = -\text{inf} \), and all NaNs are equal to themselves, though they are incomparable with any number (for example, \( +\text{NAN}(0) \leq +\text{NAN}(0) \) is false, but \( +\text{NAN}(0) = +\text{NAN}(0) \) is true).

Numerical constants are entered and printed in HimML as \( x \) or \( x:y \), where \( x \) and \( y \) represent real numbers, possibly followed by a scale. The notation \( x:y \) denotes the complex number \( x + iy \). There can be no confusion with the \( : \) character used in type constraints, because no type may begin with a character lying at the start of a number. A real number is defined by the regular expression (in lex syntax):

\[
\begin{align*}
\text{(INT)}E\text{(INT)} \\
| \text{(INT)}\cdot\text{(DIG)}+ \\
| \text{(INT)}\cdot\text{(DIG)}+E\text{(INT)} \\
| [+\text{-}]\text{inf} \\
| [+\text{-}]\text{NAN}(\{0-7\})
\end{align*}
\]

where \( \text{DIG} = \{0-9\} \) and \( \text{INT} = (\text{\~{}?}\text{DIG})+ \), with the obvious interpretations. Infinities and NaNs cannot be entered in a non-IEEE754 system.

The exception:

\[
\text{exception Complex}
\]

is raised whenever an operation defined only on real values is applied on a complex with a non-zero imaginary part (even if it is denormalized).

The numerical functions are:

- **classify** \( ' #a \rightarrow \text{numClass} \times \text{numClass} \) classifies the real and imaginary parts of the numerical value in argument.

- **denormal** \( ' #a \rightarrow \text{bool} \) returns true if and only if its argument is zero or a denormalized number. That is, it returns true if and only if classify returns a pair of classifications, which are both \( \text{MDenormal, Zero} \) or \( \text{PDenormal} \). denormal is useful because dividing by a number is in fact legal only when this number is not denormal, in the IEEE754 representation system. In short, being denormalized is the right criterion for testing whether a number is invertible (and not comparison with \( 0 \)).

- **overflown** \( ' #a \rightarrow \text{bool} \) returns true if and only if its argument is infinite or a NaN. That is, it returns true if and only if its real part or its imaginary part is infinite or a NaN. This function is useful to detect when a computation has produced an infinite number; usually, when doing some operations on infinite numbers, like subtraction, the results are NaNs, hence the test for being either infinite or a NaN.

- **\#+** \( ' #a \times ' #a \rightarrow ' #a \) is addition. It is declared infix, left associative of precedence 6.

- **\#-** \( ' #a \times ' #a \rightarrow ' #a \) is subtraction. It is declared infix, left associative of precedence 6.

- **\#*** \( ' #a \times ' #b \rightarrow ' #a \times ' #b \) is multiplication. It is declared infix, left associative of precedence 7.

- **\#/** \( ' #a \times ' #b \rightarrow ' #a \times ' #b \) is division; it does not raise an exception when dividing by \( 0 \) or a denormalized, except on non-IEEE754 systems (the Arith exception). It is declared infix, left associative of precedence 7.
• \( #^\wedge \) : num \ast num \rightarrow num \) is exponentiation (as in FORTRAN); the typing engine knows a special case about this one: when \( n \) is a constant number and \( e \) is an expression, then \( #^\wedge \) in the expression \( e \#^\wedge n \) is given type \( \#a \ast \#a \rightarrow \#a \wedge n \). It is declared infix, right associative of precedence 8.

We could almost define \( #^\wedge \) as \( fn (a,b) \Rightarrow \exp (b \# \ast \log a) \), except for rounding mechanisms and type inference.

• \( #^\wedge \) : \'(\#a \rightarrow \#a \) is negation.

• \( pi \) : num is the famous number \( \pi \).

• \( \text{fsqr} \) : \#'a \rightarrow \#'a^2 \) squares its argument.

• \( \text{fsqrt} \) : \#'a \rightarrow \#'a^0.5 \) takes the square root of its argument. It is the principal determination of the square root, defined by \( \sqrt{r.e^{i.\theta}} = \sqrt{r}.e^{i.\theta/2} \) for \( r \geq 0, -\pi < \theta \leq \pi \).

• \( \text{fabs} \) : \#'a \rightarrow \#'a \) computes the norm of its argument. The norm of \( z \) is \( |z| = \sqrt{z.\overline{z}} \), where \( \overline{z} \) is the complex conjugate of \( z \). When \( z \) is real, the norm of \( z \) is simply its absolute value.

• \( \text{conj} \) : \#'a \rightarrow \#'a \) computes the conjugate of a complex numerical value. The complex conjugate of \( x + i.y \) is \( x - i.y \). When \( z \) is real, the conjugate of \( z \) is \( z \) itself.

• \( \text{re} \) : \#'a \rightarrow \#'a \) computes the real part of its argument. It is always a real.

• \( \text{im} \) : \#'a \rightarrow \#'a \) computes the imaginary part of its argument. It is always a real.

• \( \#< \) : \#'a \ast \#'a \rightarrow \text{bool} \) returns true if its first argument is less than its second argument, considered as real quantities. The exception Complex is raised if one of the numbers was complex. Note that comparison of NaNs may yield surprising results. For quantities with dimensions, the arguments are scaled to the default scale of the dimension \( \#'a \), and then compared. Confusing results can ensue if you are using negative scaling factors.

• \( \#> \) : \#'a \ast \#'a \rightarrow \text{bool} \) returns true if its first argument is greater than its second argument, considered as real quantities. The exception Complex is raised if one of the numbers was complex. Note that comparison of NaNs may yield surprising results. For quantities with dimensions, the arguments are scaled to the default scale of the dimension \( \#'a \), and then compared. Confusing results can ensue if you are using negative scaling factors.

• \( \#\leq \) : \#'a \ast \#'a \rightarrow \text{bool} \) returns true if its first argument is less than or equal to its second argument, considered as real quantities. The exception Complex is raised if one of the numbers was complex. Note that comparison of NaNs may yield surprising results. For quantities with dimensions, the arguments are scaled to the default scale of the dimension \( \#'a \), and then compared. Confusing results can ensue if you are using negative scaling factors.

• \( \#\geq \) : \#'a \ast \#'a \rightarrow \text{bool} \) returns true if its first argument is greater than or equal to its second argument, considered as real quantities. The exception Complex is raised if one of the numbers was complex. Note that comparison of NaNs may yield surprising results. For quantities with dimensions, the arguments are scaled to the default scale of the dimension \( \#'a \), and then compared. Confusing results can ensue if you are using negative scaling factors.

• \( \text{fmin} \) : \#'a \ast \#'a \rightarrow \#'a \) returns the least of its two arguments, considered as real quantities. The exception Complex is raised if one of the numbers was complex. Comparison of NaNs may give some surprises, as well as the use of negative scaling factors. (See <.)

• \( \text{fmax} \) : \#'a \ast \#'a \rightarrow \#'a \) returns the greatest of its two arguments, considered as real quantities. The exception Complex is raised if one of the numbers was complex. Comparison of NaNs may give some surprises, as well as the use of negative scaling factors. (See >.)
• **exp : num → num** is the exponential, or natural anti-logarithm.

• **log : num → num** is the principal determination of the natural logarithm (base $e$). In the complex plane, $\log(r.e^{i\theta}) = \log r + i\theta$ for $r \geq 0$ and $-\pi < \theta \leq \pi$.

• **exp1 : num → num** is almost syntactic sugar for $\text{fn x => exp x - 1.0}$, but gives precise results even when the argument tends to zero.

• **log1 : num → num** is almost syntactic sugar for $\text{fn x => log (1.0 + x)}$, but gives precise results even when the argument tends to zero.

• **sin : num → num** is the (trigonometric) sine. The default angle scale is the radian (defined as $\text{scale 1'r = 1}$), but of course the scale system may be used to input angles in any other unit.

• **cos : num → num** is the (trigonometric) cosine. The default angle scale is the radian.

• **tan : num → num** is the (trigonometric) tangent, quotient of sine by cosine.

• **asin : num → num** is the principal determination of the arc sine. In the complex plane, $\text{asin x}$ is defined as $-i.\log(i.z + \sqrt{1 - z^2})$.

• **acos : num → num** is the principal determination of the arc cosine. In the complex plane, $\text{acos x}$ is defined as $-i.\log(z + \sqrt{z^2 - 1})$.

• **atan : num → num** is the principal determination of the arc tangent. In the complex plane, $\text{atan x}$ is defined as $i/2.\log \frac{1 + z}{1 - z}$.

• **sh : num → num** is the hyperbolic sine. Equivalently, it might be defined as $\text{fn z => 0.5 * (exp z - exp (z^-))}$, except that this would not be as precise near 0.

• **ch : num → num** is the hyperbolic cosine function. It could have been defined as $\text{fn z => 0.5 * (exp z + exp (z^-))}$.

• **th : num → num** is the hyperbolic tangent, quotient of the hyperbolic sine by the hyperbolic cosine.

• **ash : num → num** is the principal determination of the argument hyperbolic sine. We have $\text{ash z} = \log(z + \sqrt{1 + z^2})$.

• **ach : num → num** is the principal determination of the argument hyperbolic cosine. We have $\text{ach z} = \log(z + \sqrt{z^2 - 1})$.

• **ath : num → num** is the principal determination of the argument hyperbolic tangent. We have $\text{ath z} = \frac{1}{2}.\log \frac{1 + z}{1 - z}$.

• **arg : num → num** is the principal determination of the argument. This is always a real in the semi-closed interval $[-\pi, \pi]$. It could be defined as $\text{fn x => im(log x)}$.

• **floor : num → num** computes the lower integer part, or floor, of a real: this is the greatest integer less than or equal to the argument. It raises Complex if the argument is not real.

• **ceil : num → num** computes the upper integer part, or ceiling, of a real: this is the smallest integer greater than or equal to the argument. It raises Complex if the argument is not real. It is syntactic sugar for $\text{fn x => (floor (x^-))}$.

• **fdiv : '#a * '#a → num** is quotient extraction, it always returns an integer-valued real. This is syntactic sugar for $\text{fn (x,y) => floor(re(x/y))}$. It is declared infix, left associative of precedence 7.
• \texttt{fmod} : \texttt{\#a * \#a -> \#a} is the remainder operation. It is syntactic sugar for \texttt{fn (x,y) => x \#- y \#* (x fdiv y)}. It is declared infix, left associative of precedence 7.

• \texttt{fdivmod} : \texttt{\#a * \#a -> num * \#a} returns both the quotient and remainder of the numerical quantities in argument. This could be defined as \texttt{fn (x,y) => (x fdiv y, x fmod y)}. \texttt{fdivmod} is not especially quicker than computing \texttt{fdiv} and \texttt{fmod} separately, because the three functions maintain a common one-entry cache for quotients and remainders.

• \texttt{ldexp} : \texttt{\#a * int -> \#a} applied to \texttt{x} and \texttt{n} computes \texttt{x.2^n}.

• \texttt{frexp} : \texttt{num -> num * int} applied to a real number \texttt{x \neq 0} returns a real number \texttt{y} of the same sign as \texttt{x} and an integer \texttt{n} such that \texttt{0.5 \leq |y| < 1} and \texttt{x = y.2^n}. If \texttt{x = 0.0}, returns \texttt{(0,0,0)}. In general, if \texttt{x} is complex and \texttt{\Re x \neq 0}, then it returns \texttt{(y,n)} such that \texttt{0.5 \leq |\Re y| < 1} and \texttt{x = y.2^n}; if \texttt{x} is complex and \texttt{\Re x = 0}, then it returns \texttt{(x,0)}.

• \texttt{modf} : \texttt{num -> num * num} splits its real argument into integer and fractional part. If its argument is not real, then it raises \texttt{Complex}. This function might be defined as \texttt{fn x => (floor x, x \#- floor x)} on non-negative numbers, but it differs on negative numbers; e.g., \texttt{modf \~3.15} is \texttt{(\~3, \~0.15)}, not \texttt{(\~4, 0.85)}.

• \texttt{real} : \texttt{\#a -> bool} tests whether the argument is real, that is, if it has a zero imaginary part. This is the same as \texttt{fn z => im z=0.0}.

• \texttt{integer} : \texttt{num -> bool} tests whether the argument is an integer (in particular, real). This is the same as \texttt{fn z => floor(re z)=z}.

• \texttt{natural} : \texttt{num -> bool} tests whether the argument is a natural number. This is the same as \texttt{fn z => integer z andalso z\geq0.0}.

• \texttt{int} : \texttt{num -> int} returns the integer value of the number in argument, as an integer. It raises \texttt{Complex} if the argument is a complex number. If the argument is real, but not in the range \texttt{[min_int, max_int]}, the result is unspecified. If the argument is not an integer, rounding direction is unspecified.

• \texttt{num} : \texttt{int -> num} converts an integer to a real number having the same value.

• \texttt{random} : \texttt{unit -> num} draws a pseudo-random equidistributed number in the real interval \texttt{[0,1[}. The method used is based on a generalized feedback shift register [8], with generating polynomial \texttt{x^{55} + x^{24} + 1}. The period is \texttt{2^{55} - 1}, which means that, practically, the generator never loops back to any previous state.

• \texttt{zrandom} : \texttt{unit -> num} draws a pseudo-random equidistributed number in the complex square \texttt{[0,1[\times [0,1[}. The method used is the same as for \texttt{random}.

• \texttt{maybe} : \texttt{unit -> bool} draws a pseudo-random equidistributed boolean. The method used is the same as for \texttt{random} and \texttt{zrandom}, and is quicker than the semantically equivalent \texttt{fn () => random() \geq 0.5}.

### 4.8 Large Integer Arithmetic

HimML includes a port of Arjen Lenstra’s large integer package, which provides arbitrary precision arithmetic, i.e., arithmetic over a type \texttt{Int} that represents actual integers, without any size limitation as with the \texttt{int} type.

The exception:

\texttt{exception Lip of int}

is raised whenever an error occurs in any of the functions of this section. The numbers as arguments to the \texttt{Lip} exception are as follows:

45
• 1: division by zero;
• 2: modulus is zero in modular arithmetic functions;
• 3: bad modulus in Montgomery arithmetic functions—this means the modulus is 0, negative or positive but even;
• 4: Montgomery arithmetic modulus is undefined—use zmontstart to define it first;
• 5: moduli in the chinese remaindering function zchirem are not coprime;
• 6: a function was called that expected a positive argument, and got a zero or negative argument; this can be raised with zln, zsqrt, and a few other functions like zchirem;
• 7: zbezout was called with both arguments zero, or zroot was called to compute an nth root with n = 0;
• 8: a bug occurred in zbezout, please report it to goubault@lsv.ens-cachan.fr, see MAINTENANCE at the end of the OPTIONS file;
• 9: zchirem was called with identical moduli but different remainders;
• 10: an exponentiation function was called with a negative exponent;
• 11: the square root or some nth root with n even of a negative number was attempted;
• 12: a bug occurred in zpollardrho, please report it to goubault@lsv.ens-cachan.fr, see MAINTENANCE at the end of the OPTIONS file;
• 13: the second argument to zjacobi or zs jacobi is even;
• 14: zrandompprime was given a non-positive value for q;
• 15: a bug occurred in zecm, please report it to goubault@lsv.ens-cachan.fr, see MAINTENANCE at the end of the OPTIONS file;
• 16: a wrong base was supplied to one of the functions converting integers to a list of digits (e.g., zstobas); a wrong base is one that is < 2.
• 17: too many small primes were requested.
• 18: one of the random prime generator functions failed.
• 19: zecm failed to factor the input argument, but did not estimate it was prime with any sufficiently high probability.

The functions are classified into several subgroups.

Conversions

• Int : int -> Int converts a machine integer to an actual integer.
• zint : Int -> int converts an actual integer to a machine integer. In case of overflow, no exception is raised: on machines with integers in two’s complement arithmetic on k bits (which is most common), zint n the unique m, \(-2^{k-1} \leq m < 2^{k-1}\), such that \(n = m \mod 2^k\).
• znum : Int -> num converts an integer to an approximate floating-point value. Overflows are not checked, and may yield unspecified results.
• zsbastoz : int * int list -> Int converts a list of digits in a given base to an integer. More precisely, zsbastoz \((b, [a_n, \ldots, a_1, a_0]) = a_n b^n + \ldots + a_1 b + a_0\).
Large Integer Arithmetic

- \textbf{zbastoz} : \text{Int} * \text{Int list} \rightarrow \text{Int} converts a list of digits in a given base to an integer. More precisely, \(\text{zbastoz}\ (b, [a_n, \ldots, a_1, a_0]) = a_n b^n + \ldots + a_1 b + a_0\).
- \textbf{zstosymbas} : \text{Int} * \text{Int list} \rightarrow \text{Int list} converts an integer to a list of digits in the given base. More precisely, given a base \(b\) and an integer \(k\), returns a list \([a_n, \ldots, a_1, a_0]\) such that \(0 \leq a_i < b\) for every \(i\), \(0 \leq i \leq n\), \(a_n \neq 0\) if \(k \neq 0\), and \(a_n b^n + \ldots + a_1 b + a_0 = |k|\). Converts \(k = 0\) to \(\text{nil}\), and raises Lip 16 if \(b < 2\).
- \textbf{zstosymbas} : \text{Int} * \text{Int list} \rightarrow \text{Int list} converts an integer to a list of digits in the given base. More precisely, given a base \(b\) and an integer \(k\), returns a list \([a_n, \ldots, a_1, a_0]\) such that \(|a_i| \leq b/2\) for every \(i\), \(0 \leq i \leq n\), \(a_n \neq 0\) if \(n \neq 0\), and \(a_n b^n + \ldots + a_1 b + a_0 = |k|\). Converts \(k = 0\) to \(\text{nil}\), and raises Lip 16 if \(b < 2\).
- \textbf{zsign} : \text{Int} \rightarrow \text{int} returns the sign of the argument, \(-1\) if negative, \(0\) if zero, \(1\) if positive.
- \textbf{zsr} : \text{Int} \rightarrow \text{Int} computes the sum of two large integers. This uses Karatsuba’s algorithm, which computes the product of two \(n\)-digit numbers in \(O(n^{1.58})\) steps. This is faster than naive multiplication \(O(n^2)\) steps), and in theory is slower than Toom-Cook multiplication \(O(n^{2.5} \sqrt{\log n})\) steps) or Fast Fourier Transform algorithms \(O(n \log n)\) steps). But the latter are more complex, and start to be faster in practice for rather large integers only.
- \textbf{zsr} : \text{Int} \rightarrow \text{Int} computes the product of two large integers. This uses Karatsuba’s algorithm, which computes the product of two \(n\)-digit numbers in \(O(n^{1.58})\) steps. This is faster than naive multiplication \(O(n^2)\) steps), and in theory is slower than Toom-Cook multiplication \(O(n^{2.5} \sqrt{\log n})\) steps) or Fast Fourier Transform algorithms \(O(n \log n)\) steps). But the latter are more complex, and start to be faster in practice for rather large integers only.
- \textbf{zcompare} : \text{Int} * \text{Int} \rightarrow \text{int} compares two integers, returns \(-1\) if the first is less than the second, \(0\) if they are equal, and \(1\) if the first is greater than the second.
- \textbf{zneg} : \text{Int} \rightarrow \text{Int} returns the opposite of the integer in argument.
- \textbf{zabs} : \text{Int} \rightarrow \text{Int} returns the absolute value of the argument.
- \textbf{zdivmod} : \text{Int} * \text{Int} \rightarrow \text{Int} computes the quotient and remainder of two integers given as argument. The dividend \(m\) is a large integer, while the divisor \(n\) is a machine integer. This returns \((q, r)\), where \(m = nq + r\), \(|r| < |n|\), and the sign of \(r\) is the same as \(n\). Note that this is not the same condition as for \text{divmod}.

47
• \texttt{zdivmod} : \texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int} \times \texttt{Int} computes the remainder of two integers given as argument. The dividend \( m \) is a large integer, while the divisor \( n \) is a machine integer. This returns \( r \), where \( m = nq + r, |r| < |n| \), and the sign of \( r \) is the same as \( n \). Note that this is not the same condition as for \texttt{mod}. Raises \texttt{Lip 1} if \( n = 0 \).

• \texttt{zdivmod} : \texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int} \times \texttt{Int} computes the quotient and remainder of two integers given as argument. The dividend \( m \) and the divisor \( n \) are large integers. This returns \( (q, r) \), where \( m = nq + r, |r| < |n| \), and the sign of \( r \) is the same as \( n \). Note that this is not the same condition as for \texttt{divmod}. Raises \texttt{Lip 1} if \( n = 0 \).

• \texttt{zmod} : \texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int} computes the remainder of two integers given as argument. The dividend \( m \) and the divisor \( n \) are large integers. This returns \( r \), where \( m = nq + r, |r| < |n| \), and the sign of \( r \) is the same as \( n \). Note that this is not the same condition as for \texttt{mod}. Raises \texttt{Lip 1} if \( n = 0 \).

• \texttt{zexp} : \texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int}, applied to \((a, e)\), computes \( a^e \); note that, although \( a \) is a large integer, \( e \) is a machine integer. Raises \texttt{Lip 10} if \( e < 0 \) and \(|a| \neq 1 \). Note that this function may take some time and build a huge result: time and space is \( O(e \log a) \), i.e., proportional to the size of \( a \) times \( e \) itself, that is, an exponential of its size. By convention, \( 0^0 = 1 \).

• \texttt{zexp} : \texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int}, applied to \((a, e)\), computes \( a^e \); both \( a \) and \( e \) are large integers. Raises \texttt{Lip 10} if \( e < 0 \) and \(|a| \neq 1 \). Note that this function may take some time and build a huge result: time and space is \( O(e \log a) \), i.e., proportional to the size of \( a \) times \( e \) itself, that is, an exponential of its size. In short, although \( e \) is a large integer, it should not be too large for \texttt{zexp} to return at all. By convention, \( 0^0 = 1 \).

• \texttt{zrsi} : \texttt{Int} \rightarrow \texttt{Int} multiplies the argument by 2, that is, shifts it left 1 bit.

• \texttt{zasr} : \texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int}, called on \((a, n)\), shifts \( a \) left \( n \) bits (if \( n \geq 0 \)), or right \( -n \) bits (if \( n < 0 \)). That is, it returns the greatest integer in absolute value that is less in absolute value than \( a2^n \) (this is exactly \( a2^n \) unless \( n < 0 \)).

• \texttt{zasr} : \texttt{Int} \rightarrow \texttt{Int} divides the argument by 2, that is, shifts it right 1 bit. For negative arguments, this does not work as \texttt{zdivmod}, rather as \texttt{divmod}; this does not work as \texttt{asr} either. That is, while \texttt{asr} \((−3, 1)\) returns \(−2\), \texttt{zasr} \((\texttt{Int} −\texttt{3})\) returns \(−1\).

• \texttt{zasr} : \texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int}, called on \((a, n)\), shifts \( a \) right \( n \) bits (if \( n \geq 0 \)), or left \( −n \) bits (if \( n < 0 \)). That is, it returns the greatest integer in absolute value that is less in absolute value than \( a2^n \).

• \texttt{zodd} : \texttt{Int} \rightarrow \texttt{bool} returns \texttt{true} if and only if the argument is odd. Testing \texttt{zodd \((n)\)} is equivalent to \texttt{zbit \((n, 0)\)}.

• \texttt{zmakeodd} : \texttt{Int} \rightarrow \texttt{int} \times \texttt{Int} returns \((k, a)\) with \( k \) highest such that the argument equals \( 2^k.a \). Note that \( k \) is a machine integer, while \( a \) is a large integer. Raises \texttt{Lip 7} if argument is 0.

• \texttt{sqrt} : \texttt{int} \rightarrow \texttt{int} returns the floor of the square root of the argument, which is a machine integer. Raises \texttt{Lip 11} if argument is negative.

• \texttt{zsqrt} : \texttt{Int} \rightarrow \texttt{Int} \times \texttt{Int} returns \((r, d)\), where \( r \) is the floor of the square root of the argument \( a \), and \( d \) is the remainder \( a − r^2 \). Raises \texttt{Lip 11} if argument is negative.

• \texttt{zroot} : \texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int} returns the floor of the \( n \)th root of \( a \), where \((a, n)\) is provided as argument. If \( a \) is negative and \( n \) is odd, returns the opposite of the \( n \)th root of \( −a \). Raises \texttt{Lip 7} if \( n = 0 \), \texttt{Lip 11} if \( a \) is negative but \( n \) is even, \texttt{Lip 1} if \( a = 0 \) and \( n < 0 \).

• \texttt{zlog} : \texttt{Int} \rightarrow \texttt{num} returns the natural logarithm of the argument, or at least a good approximation (on 32-bit architectures, if computes it from the upper 60 bits, for a 56-bit result). If the argument \( x \) is negative, the principal branch of the logarithm is chosen, and \( \log(−x) + i\pi \) is returned. If \( x = 0 \), then \texttt{Lip 7} is raised.

48
A set of bits whose complementary is finite, those that are set in the two’s complement binary representation of the integer; while every negative integer represents a cofinite set of bits (i.e., $m = x^2 \mod p$ for some $x$), otherwise it is $-1$.

This implies that in general, $z\text{Jacobi} (m, n)$ returns $0$ if $m$ and $n$ are not coprime, and if it returns $-1$ then $m$ is not a quadratic residue modulo $n$.

Computation is based on the identities (where $n > 0$): $\left( \frac{2}{n} \right) = 0$; $\left( \frac{1}{n} \right) = 1$; $\left( \frac{-m}{n} \right) = (-1)^{(m-1)/2} \left( \frac{m}{n} \right)$ when $n$ is odd; $\left( \frac{2m}{n} \right) = (-1)^{(n^2-1)/8} \left( \frac{m}{n} \right)$ when $n$ is odd; $\left( \frac{m}{n} \right) = (-1)^{(m-1)(n-1)/4} \left( \frac{m \mod n}{n} \right)$ when both $m$ and $n$ are odd.

Raises Lip 6 if $n \leq 0$, Lip 13 if $n$ is not odd.

$z\text{Jacobi} : \text{Int} * \text{Int} \to \text{Int}$ computes the Jacobi symbol of the integers in argument. See $z\text{Jacobi}$ for details.

**Bit Manipulation**

Large integers also serve as bit vectors, of varying size. Every non-negative integer represents a finite set of bits, those that are set in the binary representation of the integer; while every negative integer represents a cofinite set of bits (i.e., a set of bits whose complementary is finite), those that are set in the two’s complement binary representation of the integer. For example, 23 is ...00010111 in binary, and represents the set \{0, 1, 2, 4\}, since $23 = 2^3 + 2^2 + 2^1 + 2^0$. And $-23$ is ...11101001, and represents the set \{0, 3, 5, 6, 7, ...\}.

- $z\text{snbits} : \text{Int} \to \text{Int}$ returns the number of bits needed to represent the argument $x$, which is a machine integer. This is $\lceil \log_2(|x| + 1) \rceil$, where $\log_2$ is base 2 logarithm.

- $z\text{nbits} : \text{Int} \to \text{Int}$ returns the number of bits needed to represent the argument $x$, which is a large integer. This is $\lceil \log_2(|x| + 1) \rceil$, where $\log_2$ is base 2 logarithm.

- $z\text{weight} : \text{Int} \to \text{Int}$ returns the number of bits set to one in the binary representation of the machine integer in argument. When the argument is negative, this is also the number of 0 bits in the binary negation of the argument (see $b\text{not}$), on two’s complement machines.

- $z\text{weight} : \text{Int} \to \text{Int}$ returns the number of bits set to one in the binary representation of $|x|$, where $x$ is the large integer in argument. In particular, for negative $n$, $z\text{weight} (\text{Int} \ n)$ does not return the same number as $z\text{weight} \ n$.

- $z\text{lowbits} : \text{Int} * \text{Int} \to \text{Int}$, applied to the large integer $m$ and the machine integer $n$, returns the $n$ lowest bits of $m$ as a large integer. Works as though $m$ is in two’s complement, and always returns a non-negative number.

- $z\text{highbits} : \text{Int} * \text{Int} \to \text{Int}$, applied to the large integer $m$ and the machine integer $n$, returns the $n$ highest bits of $|m|$ as a large integer. $z\text{highbits} \ (m, n)$ is equivalent to $z\text{asar} \ (|m|, z\text{snbits} \ (m) - n)$ if $z\text{snbits} \ (m) > n$, $|m|$ otherwise.

- $z\text{cat} : \text{Int} * \text{Int} \to \text{Int}$ concatenates the bit vectors in argument. Calling $z\text{cat} \ (m, n)$ is equivalent to $z\text{or} \ (z\text{asl} \ (m, z\text{nbits} \ (n)), n)$. Concatenating two bit vectors $m$ and $n$, where $n$ is instead known to contain $k$ significant bits, should instead be done by calling $z\text{or} \ (z\text{asl} \ (m, k), n)$.

- $z\text{reverse} : \text{Int} * \text{Int} \to \text{Int}$ returns the bit reversal of the machine integer $m$ in first argument. Second argument $k$ is the number of bits in $m$ that should be considered, and is truncated to 0 if negative, and to the number of bits in a machine integer if greater.
• \text{zreverse} : \text{Int} \times \text{int} \rightarrow \text{Int} \text{ returns the bit reversal of the large integer } m \text{ in first argument. The second argument } k \text{ is the number of bits in } m \text{ that should be considered, and is truncated to } 0 \text{ if negative. The integer } m \text{ is viewed in two’s complement, and even if } m < 0, \text{zreverse} (m, k) \text{ is } \geq 0. \text{ Just reversing } m \text{ is done by calling \text{zreverse} (m, znbits (m)), provided } m \geq 0.

• \text{setofInt} : \text{Int} \rightarrow \text{int set} \text{ returns the set of bits set to 1 in the integer argument. Raises Lip 10 if the argument is negative, in which case the set would be infinite and hence not representable of type \text{int set}.

• \text{Intofset} : (\text{int} \rightarrow 'a) \rightarrow \text{Int} \text{ computes the greatest common divisor of the arguments by the classical Euclidean algorithm. This might be faster than \text{gcd} in special cases. Raises Lip 7 if both arguments are 0.}

• \text{zgcd} : \text{Int} \times \text{Int} \rightarrow \text{Int} \text{ computes the greatest common divisor of the arguments, by the binary method. Raises Lip 7 if both arguments are 0.}

• \text{zgcdce} : \text{Int} \times \text{Int} \rightarrow \text{Int} \text{ computes the greatest common divisor of the arguments, by the classical Euclidean algorithm. This might be faster than \text{zgcd} in special cases. Raises Lip 7 if both arguments are 0.}

• \text{zbezout} : \text{Int} \times \text{Int} \rightarrow (\text{Int} \times \text{Int}) \times \text{Int} \text{ returns such that } a.xa + b.xb = d. \text{ This is also sometimes called the extended Euclidean algorithm. The gcd is always } \geq 1.

This raises Lip 7 if both \(a\) and \(b\) are zero, in which case the gcd is undefined.

This might raise Lip 8, but should not. If this happens, this is a bug, and this should be reported to the author goubault@lsv.ens-cachan.fr, see MAINTENANCE at the end of the OPTIONS file.

• \text{zchirem} : \text{Int} \times \text{Int} \times \text{Int} \times \text{Int} \rightarrow \text{Int} \text{ applied to } (ma, a, mb, b), \text{ returns } d \text{ such that } d = a \text{ modulo } ma, \text{ and } d = b \text{ modulo } mb, \text{ and } 0 \leq d < ma.mb—unless } ma = mb, \text{ in which case } a \text{ and } b \text{ should be equal, and } d = a = b.

Raises Lip 6 if \(ma \leq 0\) or \(mb \leq 0\). Raises Lip 9 if \(ma = mb\) but \(a \neq b\), and Lip 5 if \(ma \neq mb\) but \(ma\) and \(mb\) are not coprime.

Euclidean Algorithms

• \text{zgcd} : \text{Int} \times \text{Int} \rightarrow \text{Int} \text{ computes the greatest common divisor of the arguments by the binary method. Raises Lip 7 if both arguments are 0.}

• \text{zgcdce} : \text{Int} \times \text{Int} \rightarrow \text{Int} \text{ computes the greatest common divisor of the arguments, by the classical Euclidean algorithm. This might be faster than \text{zgcd} in special cases. Raises Lip 7 if both arguments are 0.}

• \text{zbezout} : \text{Int} \times \text{Int} \rightarrow (\text{Int} \times \text{Int}) \times \text{Int} \text{ computes the gcd } d \text{ of } a \text{ and } b, \text{ and two coefficients } xa, xb \text{ as in Bezout’s Theorem, i.e., such that } a.xa + b.xb = d. \text{ This is also sometimes called the extended Euclidean algorithm. The gcd is always } \geq 1.

This raises Lip 7 if both \(a\) and \(b\) are zero, in which case the gcd is undefined.

This might raise Lip 8, but should not. If this happens, this is a bug, and this should be reported to the author goubault@lsv.ens-cachan.fr, see MAINTENANCE at the end of the OPTIONS file.

• \text{zchirem} : \text{Int} \times \text{Int} \times \text{Int} \times \text{Int} \rightarrow \text{Int} \text{ applied to } (ma, a, mb, b), \text{ returns } d \text{ such that } d = a \text{ modulo } ma, \text{ and } d = b \text{ modulo } mb, \text{ and } 0 \leq d < ma.mb—unless } ma = mb, \text{ in which case } a \text{ and } b \text{ should be equal, and } d = a = b.

Raises Lip 6 if \(ma \leq 0\) or \(mb \leq 0\). Raises Lip 9 if \(ma = mb\) but \(a \neq b\), and Lip 5 if \(ma \neq mb\) but \(ma\) and \(mb\) are not coprime.
Standard Modular Arithmetic

- **zmadd**: \( \text{Int} \times \text{Int} \times \text{Int} \to \text{Int} \) adds modulo \( m \): \( \text{zmadd} \ (a, b, m) \) computes \((a + b) \mod m \). It is assumed that \( 0 \leq a, b < m \); result is also \( 0 \leq \) and \( < m \). Produces erratic results otherwise. Raises Lip 2 if \( m = 0 \).

- **zmsub**: \( \text{Int} \times \text{Int} \times \text{Int} \to \text{Int} \) subtracts modulo \( m \): \( \text{zmsub} \ (a, b, m) \) computes \((a - b) \mod m \). It is assumed that \( 0 \leq a, b < m \); result is also \( 0 \leq \) and \( < m \). Produces erratic results otherwise. Raises Lip 2 if \( m = 0 \).

- **zmmul**: \( \text{Int} \times \text{Int} \times \text{Int} \to \text{Int} \) multiplies modulo \( m \): \( \text{zmmul} \ (a, b, m) \) computes \(ab \mod m \). All arguments are large integers. It is assumed that \( 0 \leq a, b < m \); result is also \( 0 \leq \) and \( < m \). Produces erratic results otherwise. Raises Lip 2 if \( m = 0 \).

- **zmsqr**: \( \text{Int} \times \text{Int} \to \text{Int} \) computes squares modulo \( m \): \( \text{zmsqr} \ (a, m) \) computes \(a^2 \mod m \). It is assumed that \( 0 \leq a < m \); result is also \( 0 \leq \) and \( < m \). Produces erratic results otherwise. Raises Lip 2 if \( m = 0 \).

- **zmdiv**: \( \text{Int} \times \text{Int} \times \text{Int} \to \text{Int} \) divides modulo \( m \): \( \text{zmdiv} \ (a, b, m) \) computes \(a/b \mod m \). It is assumed that \( 0 \leq a, b < m \); result is also \( 0 \leq \) and \( < m \). Produces erratic results otherwise. Raises Lip 2 if \( m = 0 \) or \( b = 0 \). If \( m \) is not coprime with \( b \), it might also be the base that the quotient is undefined, in which case no exception is raised, rather some factor of \( m \) is returned.

- **zminv**: \( \text{Int} \times \text{Int} \to \text{Int} \) computes inverses modulo \( m \): \( \text{zminv} \ (a, m) \) computes \(a^{-1} \mod m \). It is assumed that \( 0 \leq a < m \); result is also \( 0 \leq \) and \( < m \). Produces erratic results otherwise. Raises Lip 2 if \( m = 0 \) or \( a = 0 \). It might also be the base that the inverse is undefined, in which case no exception is raised, rather some factor of \( m \) is returned—namely \( gcd(a, m) \).

- **zmsexp**: \( \text{Int} \times \text{Int} \times \text{Int} \to \text{Int} \), applied to \((a, e, m)\), computes \(a^e \mod m \). All arguments are machine integers. Raises Lip 2 if \( m = 0 \). By convention, \( 0^0 = 1 \).

- **zsmexp**: \( \text{Int} \times \text{Int} \times \text{Int} \to \text{Int} \), applied to \((a, e, m)\), computes \(a^e \mod m \). The exponent \( e \) is a machine integer; however, this is not faster than calling \( \text{zmexp} \), since \( \text{zsmexp} \) calls \( \text{zmexp} \). Raises Lip 2 if \( m = 0 \), Lip 10 if \( e \) is negative and \( a \) and \( m \) are not coprime. By convention, \( 0^0 = 1 \).

- **zmexp**: \( \text{Int} \times \text{Int} \times \text{Int} \to \text{Int} \), applied to \((a, e, m)\), computes \(a^e \mod m \). The exponent \( e \) is a large integer. Raises Lip 2 if \( m = 0 \), Lip 10 if \( e \) is negative and \( a \) and \( m \) are not coprime. By convention, \( 0^0 = 1 \).

- **zmsexpm**: \( \text{Int} \times \text{Int} \times \text{Int} \to \text{Int} \), applied to \((a, e, m)\), computes \(a^e \mod m \), just like \( \text{zmexp} \). However, it uses the \( m \)-ary method, which is faster than \( \text{zmexp} \) if \( a \) is not too small. Raises Lip 2 if \( m = 0 \), Lip 10 if \( e \) is negative and \( a \) and \( m \) are not coprime. By convention, \( 0^0 = 1 \).

- **zm2exp**: \( \text{Int} \times \text{Int} \to \text{Int} \), applied to \((e, m)\), computes \(2^e \mod m \). Raises Lip 2 if \( m = 0 \), Lip 10 if \( e \) is negative and \( m \) is even.

- **zmexp2**: \( \text{Int} \times \text{Int} \times \text{Int} \times \text{Int} \times \text{Int} \to \text{Int} \), called on \((a_1, e_1, a_2, e_2, m)\), computes \(a_1^{e_1} a_2^{e_2} \mod m \). Raises Lip 2 if \( m = 0 \), Lip 10 if \( e_1 \) or \( e_2 \) is negative. This uses Shamir’s method with sliding window of size 1, 2 or 3, depending on the maximal size of \( e_1 \) or \( e_2 \). This should be used if \( e_1 \) and \( e_2 \) are approximately of the same size.
Montgomery Modular Arithmetic

Modular multiplications can be done division free and therefore somewhat faster (about 20%), if the Montgomery representation is used [10]. Converting to and from Montgomery representation takes one Montgomery multiplication each, so it only pays to use Montgomery representation if many multiplications have to be carried out modulo a fixed odd modulus.

To use Montgomery arithmetic, first initialize the modulus \( N \) by using \texttt{zmontstart}, and convert all operands to their Montgomery representation by \texttt{ztom}, but do not convert exponents. Use the addition, subtraction, multiplication, squaring, division, inversion, and exponentiation functions, which all start with \texttt{zmont}, below on the converted operands, just as you would use the ordinary modular functions (starting with \texttt{zm}). The results can be converted back from Montgomery representation to ordinary numbers modulo \( N \) using \texttt{zmonttoz}.

- \texttt{zmontstart : Int -> unit} initializes Montgomery arithmetic mod \( N \), the argument. Raises \texttt{Lip 3} if \( N \) is not both positive and odd.
  
  If no exception is raised, all subsequent computations using Montgomery arithmetic will use modulus \( N \). Mixing ordinary large integers and Montgomery numbers, or Montgomery numbers based on different moduli yields surprising results; this should be left to experimented users only.

- \texttt{ztomont : Int -> Int} converts an ordinary large integer to the corresponding Montgomery number mod \( N \). Raises \texttt{Lip 3} if \( N \) is undefined.

- \texttt{zmonttoz : Int -> Int} converts a Montgomery number mod \( N \) to the corresponding ordinary large integer. Raises \texttt{Lip 3} if \( N \) is undefined.

- \texttt{zmontadd : Int * Int -> Int} adds two Montgomery numbers mod \( N \). This actually just calls \texttt{zmadd} with \( N \) as modulus. Raises \texttt{Lip 3} if \( N \) is undefined.

- \texttt{zmontsub : Int * Int -> Int} subtracts two Montgomery numbers mod \( N \). This actually just calls \texttt{zmsub} with \( N \) as modulus. Raises \texttt{Lip 3} if \( N \) is undefined.

- \texttt{zsmontmul : Int * int -> Int} takes one Montgomery number mod \( N \) and an ordinary machine integer, multiplies them and returns the result as a Montgomery number mod \( N \). This actually just calls \texttt{zsmmul} with \( N \) as modulus. Raises \texttt{Lip 3} if \( N \) is undefined.

- \texttt{zmontmul : Int * Int -> Int} multiplies two Montgomery numbers mod \( N \). This is essentially the only function, apart from \texttt{zmontsqr} and \texttt{zmontinv}, that justifies using Montgomery arithmetic. Raises \texttt{Lip 3} if \( N \) is undefined.

- \texttt{zmontsqr : Int -> Int} squares the Montgomery number mod \( N \) in argument. This is essentially the only function, apart from \texttt{zmontmul} and \texttt{zmontinv}, that justifies using Montgomery arithmetic. Raises \texttt{Lip 3} if \( N \) is undefined.

- \texttt{zmontdiv : Int * Int -> Int} divides two Montgomery numbers mod \( N \). Raises \texttt{Lip 3} if \( N \) is undefined, \texttt{Lip 1} if second argument is 0 mod \( N \). If \( N \) is not coprime with the second argument, it might also be the case that the quotient is undefined. In this case, no exception is raised, rather some factor of \( N \) is returned (as a regular integer, \textit{not} as a Montgomery number).

- \texttt{zmontinv : Int -> Int} computes the inverse of a Montgomery number mod \( N \). Raises \texttt{Lip 3} if \( N \) is undefined, \texttt{Lip 1} if argument is 0 mod \( N \). If \( N \) is not coprime with the argument, it might also be the case that the inverse is undefined. In this case, no exception is raised, rather some factor of \( N \) is returned (as a regular integer, \textit{not} as a Montgomery number).

This is one of the rare functions, apart from \texttt{zmontmul} and \texttt{zmontsqr}, that justifies using Montgomery arithmetic. Indeed, computing the inverse mod \( N \) in Montgomery representation means doing just one Montgomery multiplication by a constant.
• \texttt{zmontexp} : Int * Int -> Int applied to \((a, e)\), computes \(a^e \mod N\). Although \(a\) is a Montgomery number, \(e\) is not, and is a regular large integer. Raises Lip 3 if \(N\) is undefined, Lip 10 if \(e\) is negative and \(a\) and \(N\) are not coprime. By convention, \(0^0 = 1\).

• \texttt{zmontexpm} : Int * Int -> Int, called on \((a, e)\), computes \(a^e \mod N\), just like \texttt{zmontexp}. However, it uses the \textit{m}-ary method, which is faster than \texttt{zmontexp} if \(a\) is not too small. Raises Lip 3 if \(N\) is undefined, Lip 10 if \(e\) is negative and \(a\) and \(N\) are not coprime. By convention, \(0^0 = 1\).

• \texttt{zmontexp2} : Int * Int * Int * Int -> Int, called on \((a_1, e_1, a_2, e_2)\), computes \(a_1^{e_1} \cdot a_2^{e_2} \mod N\). Raises Lip 2 if \(m = 0\), Lip 10 if \(e_1\) or \(e_2\) is negative. This uses Shamir’s method with sliding window of size 1, 2 or 3, depending on the maximal size of \(e_1\) or \(e_2\). This should be used if \(e_1\) and \(e_2\) are approximately of the same size.

Primes, Factoring

• \texttt{primes} : int -> unit -> int is a small prime enumerator generator. That is, first call \texttt{primes} \(n\) for some integer \(n\); this returns a function, call it \texttt{nextprime}. Then calling \texttt{nextprime ()} repeatedly returns 2, then 3, 5, 7, 11, ..., i.e., all primes that hold in a machine integer (small primes). After \texttt{nextprime} has exhausted all small primes, raises \texttt{Lip 17}.

Note that each new call to \texttt{primes} generates a new prime enumerator. That is, just calling \texttt{primes n ()} repeatedly will just return 2 each time, by computing a new enumerator each time, and calling it only once.

The small prime number enumerator \texttt{primes} is basically an implementation of Eratosthenes’ sieve. The number \(n\) is an estimation of the largest prime you will need. It serves to allocate a table of \(m\) elements, where \(m\) is the largest integer such that \(2m(2m + 1) \leq n\). On 32 machines, calling \texttt{primes max_int} is recommended. On 64 machines, doing so will generate a huge table, which will take a few seconds just to initialize, so a smaller value of \(n\), say, 5 000 000 000, is recommended.

• \texttt{zpollardrho} : Int * int -> (Int * Int) option is an implementation of Pollard’s \(\rho\) algorithm. \texttt{zpollardrho \((n, k)\)} tries to factor \(n\), in \(k\) iterations or less. The value \(k = 0\) is special and is meant to denote no bound on the number of iterations.

If the algorithm succeeds, it returns \texttt{SOME \((r, c)\)}, where \(r\) is a non-trivial factor of \(n\), \(c = n/r\) and \(r \leq c\). If \(n < 0\), returns \texttt{SOME \((-1, n)\)}. If the algorithm fails, returns \texttt{NONE}.

 arisen, this should be reported to the author goubault@lsv.ens-cachan.fr, see MAINTENANCE at the end of the OPTIONS file.

• \texttt{ztrydiv} : Int * int * int -> (Int * Int) option applies to a large integer \(n\), and two machine integers \(a\) and \(b\), and computes the smallest (positive) prime divisor \(p\) of \(n\) that is \(\geq a\) and \(\leq b\). If \(p\) exists, it returns \texttt{SOME \((p, n/p)\)}, otherwise \texttt{NONE}.

This works by calling \texttt{primes \(b\)} to get a function that enumerates all small primes. All primes \(< a\) are first discarded, then all primes between \(a\) and \(b\) are tested. This is only intended for large \(n\), as it does not stop as soon as it goes past \(\sqrt{n}\), for small \(a\), as enumerating all primes \(< a\) takes some time for large \(a\), and for \(b\) not too large (see remark on the efficiency of \texttt{primes} on 64 bit machines).

• \texttt{zprime} : Int * int -> bool takes a large integer \(m\) and a machine integer \(k\), and tests whether \(m\) is prime. This runs a probabilistic tests for at most \(k + 1\) runs. It first uses \texttt{ztrydiv} to find small factors first. If none is found, the Miller-Rabin test is run \(k + 1\) times: when \texttt{zprime} returns \texttt{false}, then \(m\) is definitely not prime, otherwise it is prime with probability at least \(1 - (1/4)^{k+1}\).

• \texttt{zrandl} : int * (Int -> Int) -> Int draws an equidistributed large integer of \(|k|\) bits, where \(k\) is the first argument. If \(k < 0\), returns opposites of such numbers. The second argument is a pseudo-random generator function, like \texttt{irand}, which takes a bound \(x\) and returns an equidistributed random integer in \([0, x]\).
Using \texttt{irand} is not recommended for cryptographic applications, instead cryptographically secure pseudo-random number generators should be used.

The second argument should preferably be a function that does not raise any exception, otherwise memory leaks may occur.

- \texttt{zrandl1 : int -> Int} is equivalent to \texttt{fn nbits => zrandl (nbits, irand)}, but faster.

- \texttt{zrand : Int * (int -> int) -> Int} draws an equidistributed large integer of absolute value $\lt m$, where $m$ is the first argument (or returns 0 if $m = 0$), and of the same sign as $m$. The second argument is a pseudo-random generator function, like \texttt{irand}, which takes a bound $x$ and returns an equidistributed random integer in $[0,x]$. Using \texttt{irand} is not recommended for cryptographic applications, instead cryptographically secure pseudo-random number generators should be used.

The second argument should preferably be a function that does not raise any exception, otherwise memory leaks may occur.

- \texttt{zrandl1 : Int -> Int} is equivalent to \texttt{fn nbits => zrandl (nbits, irand)}, but faster.

- \texttt{zrandprime : int * int * (int -> int) -> Int} applied to $k$, $n$ and $f$, draws random primes of $n$ bits. This calls $f$ to draw random numbers, which are then tested for primality.

  More precisely, if $|n| \geq 2$, then this returns a random probable prime of $|n|$ bits, where prime testing is done by calling \texttt{zprime} with argument $k$; if $|n| < 2$, then it raises Lip 18. If $n < 0$, in addition, the returned prime number is congruent to 3 modulo 4.

  This works by picking odd numbers of the right size, keeping adding two until it is probably prime; or it is too large, in which case we pick again, and start adding again, in accordance with NIST DSS Appendix. Using $f = \texttt{irand}$ is not recommended for cryptographic applications, instead cryptographically secure pseudo-random number generators should be used.

  The third argument should preferably be a function that does not raise any exception, otherwise memory leaks may occur.

- \texttt{zrandprime1 : int * int -> Int} is equivalent to \texttt{fn (nbits, ntries) => zrandprime (nbits, ntries, irand)}.

- \texttt{zrandpprime : int * int * Int * (int -> int) -> Int * Int} is used to generate a probable prime number $p$ of exactly $|k|$ bits such that $q$ divides $p - 1$, where $q$ is given. The arguments are $k$, a machine integer $n$ (used in calling \texttt{zprime}, so that $p$ is prime with probability $\geq 1 - (1/4)^{n+1}$), the large integer $q$, and a pseudo-random generator function $f$ taking a bound $x$ and returning an equidistributed random integer in $[0,x]$, e.g., \texttt{irand}. Raises Lip 14 if $q \leq 0$, Lip 18 if failed to generate $p$, typically because $q$ already uses $\geq |k|$ bits. Otherwise, returns $(p, \lfloor p/q \rfloor)$. If $k < 0$, in addition, $p$ will be congruent to 3 modulo 4.

  This works by generating $p$ at random amongst $|k|$-bit numbers such that $q$ divides $p - 1$, until $p$ is probably prime, in accordance with NIST DSS Appendix. This only works if $n$ is substantially larger than $\texttt{znbits (q)}$. Using $f = \texttt{irand}$ is not recommended for cryptographic applications, instead cryptographically secure pseudo-random number generators should be used.

  \texttt{zrandpprime} is used to generate pairs $(p,q)$ of prime numbers such that $q$ divides $p - 1$, where $p$ and $q$ have specified numbers of bits. To do this, first generate $q$ using \texttt{zrandprime}, then call \texttt{zrandpprime} to generate $p$.

  The fourth argument should preferably be a function that does not raise any exception, otherwise memory leaks may occur.

- \texttt{zrandpprime1 : int * int * Int -> Int * Int} is equivalent to
\[ \text{fn} \ (k, n, q) \Rightarrow \text{zrandprime} \ (k, n, q, \text{irand}) \]

- \text{zrandprime} : \text{int} \times \text{int} \times \text{Int} \times (\text{int} \rightarrow \text{int}) \rightarrow \text{Int} \times \text{Int} \] generates a pair of probable prime numbers \( p \) and \( q \) such that \( q \) has \( |k| \) bits and \( p = qr + 1 \), where \( k \) and \( r \) are given. The arguments are \((k, n, r, f)\), where \( n \) is given to \text{zprime} to test whether both \( p \) and \( q \) are probably prime, \( r \) is the machine integer above, and \( f \) is a pseudo-random generator taking a bound \( x \) and returning an equidistributed random integer in \([0, x]\), e.g., \text{irand}. If \( k < 0 \), in addition \( q \) will be congruent to 3 modulo 4. Raises Lip 18 if fails, typically if \( r < 2 \) or \( r \) is odd.

Using \( f = \text{irand} \) is not recommended for cryptographic applications, instead cryptographically secure pseudo-random number generators should be used.

The fourth argument should preferably be a function that does not raise any exception, otherwise memory leaks may occur.

- \text{zrandprime1} : \text{int} \times \text{int} \times \text{Int} \rightarrow \text{Int} \times \text{Int} \] is equivalent to \[ \text{fn} \ (k, n, r) \Rightarrow \text{zrandprime} \ (k, n, r, \text{irand}) \]

- \text{zrandgprime} : \text{int} \times \text{int} \times \text{bool} \times (\text{int} \rightarrow \text{int}) \rightarrow \text{Int} \times \text{Int} \times \text{Int} \] takes four arguments \( k, n, \text{first}, \text{f} \), and returns \( p, q, g \) where \( p \) and \( q \) random probable primes (tested with \text{zprime} with argument \( n \); if \( k < 0 \), in addition, \( q \) is congruent to 3 modulo 4) such that \( p = 2q + 1 \) (this uses \text{zrandprime}), and a generator \( g \) of the multiplicative group of all numbers prime to \( p \) less than \( p \). If \( \text{first} \) is \text{true}, then \( g \) will be the smallest generator, otherwise \( g \) will be selected at random. Raises Lip 18 if it fails.

Using \( f = \text{irand} \) is not recommended for cryptographic applications, instead cryptographically secure pseudo-random number generators should be used.

The fourth argument should preferably be a function that does not raise any exception, otherwise memory leaks may occur.

- \text{zrandgprime1} : \text{int} \times \text{int} \times \text{bool} \rightarrow \text{Int} \times \text{Int} \times \text{Int} \] is equivalent to \[ \text{fn} \ (k, n, \text{first}) \Rightarrow \text{zrandprime} \ (k, n, \text{first}, \text{irand}) \]

- \text{zecm} : \text{Int} \times \text{int} \times \text{int} \times \text{int} \rightarrow (\text{Int} \times \text{bool}) \text{ option}, called on \((n, \text{ncurves}, \phi_1, \text{ntries})\), tries to factor the large integer \( n \). It returns \text{NONE} if \( n \) was found to be probably prime after \( \text{ntries} \) primality tests. Otherwise, it returns \text{SOME} \((f, b)\): if \( b = \text{true} \), then \( f \) is a non-trivial factor of \( n \); if \( b = \text{false} \) (which should only happen in very exceptional circumstances), then a non-trivial factor of \( n \) can be obtained by looking at the factorization of \( f^n - f \).

This runs by using Arjen Lenstra’s elliptic curve algorithm. Up to \text{ncurves} random elliptic curves are drawn with first phase bound \( \phi_1 \) (this increases by 1.02 at each new elliptic curve).

This may raise Lip 19 if \text{zecm} fails to reach any conclusion, or if \( n \) is too small (on 32-bit machines, \( n \leq 2^{15} \approx 32768 \); in general, \( n > \sqrt{\text{max_int}} \) is never too small: in these cases, use \text{ztrydiv} first).

Raises Lip 15 if a bug occurred in \text{zecm}. In case this happens, this should be reported to the author goubault@lsv.ens-cachan.fr, see MAINTENANCE at the end of the OPTIONS file.

### 4.9 Input/Output

Two types are defined, first that of output streams \text{outstream}:

\[
\text{type } '\text{rest} \ 	ext{outstream} = [\text{[put:}\text{string} \rightarrow \text{unit}, \ldots : '\text{rest}]]
\]

means that output streams are extensible records (classes) with at least a \text{put} method, to output a string to the stream. For example, to make a stream that prints to a string stored inside a ref \text{sr:string ref}:

\[
[\text{[put = fn text } \Rightarrow \ (\text{sr } := !\text{sr }^\land \text{ text})]]
\]
The type of input streams \texttt{instream} is:

\[
\text{type } \texttt{’rest instream} := [(\text{get} : \text{int} \rightarrow \text{string}, \ldots : \texttt{’rest})]|
\]

providing at least a \texttt{get} method, that reads up to \(n\) characters from the stream, \(n\) being the number in argument. If less than \(n\) characters have been read, it usually means that an end of stream has been reached. (After the end of stream is reached, repeated calls to \texttt{get} will return the empty string.) For example, an input stream reading from a string \(s : \text{string}\), with current position stored in a ref \(\texttt{posr : num ref}\) is:

\[
|\text{get } = \text{fn } n => \text{let } \text{val } \text{goal } = !\text{posr} + n \\
\text{val } \text{newpos } = \text{(if } \text{goal} < \text{size } s \text{ then } \text{goal} \text{ else } \text{size } s) \\
\text{in} \\
\text{substr}(s,!\text{posr},\text{newpos}) \\
\text{before } (\text{posr } := \text{newpos}) \\
\text{end}|
\]

Note that if you want to read from a string, the \texttt{instring} primitive does this faster.

The following exception is defined:

\[\text{exception } \texttt{IO of int}\]

is raised when the system was unable to open a file or any other I/O-related operation; it returns as argument the error code returned by the operating system. (Note that this is OS-dependent.)

The other methods that may be present in predefined streams are:

- **flush** : \texttt{unit } \rightarrow \texttt{unit} flushes the stream, when it is buffered, emptying the buffer.
- **seek** : \texttt{int } \rightarrow \texttt{unit} sets the current position in the file to the integer in argument; if this argument is negative, it does as if it were 0, that is it goes to the beginning of the file; if it is larger than the file size, it sets the position to the end of the file.
- **advance** : \texttt{int } \rightarrow \texttt{unit} does a \texttt{seek} from the current position (given by the \texttt{tell} method).
- **seekend** : \texttt{int } \rightarrow \texttt{unit} does a \texttt{seek} from the end of file. Notice that interesting argument values in this case are negative.
- **tell** : \texttt{unit } \rightarrow \texttt{int} returns the current position in the file, starting at the beginning, for which it is 0.
- **getline** : \texttt{unit } \rightarrow \texttt{string} reads a whole line, until and including the newline character. If instead the end of file is reached, there won’t be any newline character in the resulting string. After the end of file is reached, repeated calls to \texttt{getline} will return the empty string.
- **truncate** : \texttt{unit } \rightarrow \texttt{unit} truncates a file open for writing (or for appending) at the current position. This is useful notably when seeking back into the file, and writing there, to discard any spurious data that might remain following the current position in the file.
- **close** : \texttt{unit } \rightarrow \texttt{unit} closes the current file. However, although the file is physically flushed and closed (so that, on systems with exclusive access to files, others can afterwards access the file), it will only be definitively closed when the garbage collector has determined that the associated stream was not used any more. If any method is invoked on a closed file, the file is first automatically reopened in the state it was in when it was last closed (except if it was not needed, like for the \texttt{tell} method). This way, even with strange control flows, coming for example from uses of \texttt{callcc} and \texttt{throw}, everything should go smoothly (no need to forecast when to reopen the file).

The input/output values and functions are:
• print : 'a outstream -> dynamic -> unit prints the value stored in the dynamic on the specified output stream, in the same format that is used by the toplevel loop. The type is not printed, though. Strings are printed with quotes, and with control characters escaped. If it is desired to print strings merely as the sequence of their characters, apply the put method of the stream. Try: #put stdout "Hello, world\n". (And don’t forget to #flush stdout () afterwards to see your result printed.)

• pretty : 'a outstream -> dynamic -> unit pretty-prints the value stored in the dynamic on the specified output stream, as done by the toplevel loop. The type is not printed, and the right margin is the system default (usually, 80).

• stdout : |[put:string -> unit, flush:unit -> unit]| is the standard output stream, that prints on the console if not redirected. This stream is buffered, the flush method flushes the buffer.

• stderr : |[put:string -> unit, flush:unit -> unit]| is the standard error stream; it prints on the console if not redirected. This stream is buffered, the flush method flushes the buffer (on most operating systems, an end-of-line also flushes the buffer).

• outfile : string -> |[put:string -> unit, flush:unit -> unit, seek:int -> unit, advance:int -> unit, seekend:int -> unit, tell:unit -> int, truncate:unit -> unit, close:unit -> unit]| opens the file whose name is given, creating it if it does not exist, reinitializing it to zero length, and returns an output stream. If the file could not be opened, the IO exception is raised, applied to the error code. The stream is buffered, the flush method flushes the buffer.

• appendfile : string -> |[put:string -> unit, flush:unit -> unit, seek:int -> unit, advance:int -> unit, seekend:int -> unit, tell:unit -> int, truncate:unit -> unit, close:unit -> unit]| opens the file whose name is given, creating it if it does not exist, setting the current position to the end of file, and returns an output stream. If the file could not be opened, the IO exception is raised, applied to the error code. The stream is buffered, the flush method flushes the buffer.

• outprocess : string * string list -> |[put:string -> unit, flush:unit -> unit, kill:unit -> unit]| creates a process which will execute the shell command given in argument in parallel with the current HimML process. This shell command is given in the form (command name, arguments). The command name is searched for, using the PATH shell variable, with arguments as given.

Strings can be sent to the standard input of this process by using the put method (followed by flush to really send the message), and the process can be terminated by calling the kill method.

Contrarily to files, which can be closed and then revived when necessary, a killed process cannot be revived, and an IO exception will be raised when attempting to write to a dead process.
If the process could not be created, then an IO exception is raised (normally, IO 2, “no such file or directory”).
The kill method raises an IO exception when the process exited with a non-zero exit code (as returned by the C function exit). Then, if $n$ is this code, IO $n$ is raised. (To be fair, $n$ is first taken modulo 256, and only if the result is non-zero is the exception raised, with $n$ mod 256 as argument.)
The child process can exit by itself, and this can be detected by the fact that putting a string to the child (then flushing, to be sure that the message has really been sent) will raise an IO error (normally, IO 32, “broken pipe”). It is then good policy to kill the process, as it allows the operating system to reclaim process structures allocated for the child (at least on Unix, where this is necessary).

- stdin : $\{\text{get:int \to string, getline:unit \to string}\}$ is the standard input stream, that reads from the console if not redirected. This stream is usually buffered, so that characters cannot be read until a newline character \n is typed as input.

- infile : string $\to$ $\{\text{get:int \to string, getline:unit \to string, seek:int \to unit, advance:int \to unit, seekend:int \to unit, tell:unit \to int, close:unit \to unit}\}$

  opens the file whose name is given, and returns an input stream. If the file could not be opened, the IO exception is raised, applied to the error code. The stream is buffered for speed, but no flushing method should be necessary.

- inprocess : string $\times$ string list $\to$ $\{\text{get:int \to string, getline:unit \to string, kill:unit \to unit}\}$

  creates a process which will execute the shell command given in argument in parallel with the current HimML process. This shell command is given in the form (command name, arguments). The command name is searched for, using the PATH shell variable, with arguments as given.

  All text that is printed on the parallel process’s standard output can be read by the HimML process by using the get and getline methods, and the process can be terminated by calling the kill method.

Contrarily to files, which can be closed and then revived when necessary, a killed process cannot be revived, and an IO exception will be raised when attempting to read from a dead process.

If the process could not be created, then an IO exception is raised (normally, IO 2, “no such file or directory”).
The kill method raises an IO exception when the process exited with a non-zero exit code (as returned by the C function exit). Then, if $n$ is this code, IO $n$ is raised. (To be fair, $n$ is first taken modulo 256, and only if the result is non-zero is the exception raised, with $n$ mod 256 as argument.)
The child process can exit by itself, and this can be detected by the fact that the get and getline methods will all return empty strings (end of file, at least after the internal buffer is emptied). It is then good policy to kill the process, as it allows the operating system to reclaim process structures allocated for the child (at least on Unix, where this is necessary).

- instring : string $\to$ $\{\text{get:int \to string, getline:unit \to string, seek:int \to unit, advance:int \to unit, seekend:int \to unit, tell:unit \to int}\}$

  opens a stream for reading on the string given as argument. This is a souped up version of the input stream example at the beginning of the section.
• **outstring** : string -> ![put:string -> unit,
seek:int -> unit,
advance:int -> unit,
seekend:int -> unit,
tell:unit -> int,
truncate:unit -> unit,
convert:unit -> string]

  opens a stream to write on, as if it were a file. The `convert` method is used to get the current contents of the stream in the form of a string. This is useful notably to print data to a string. The stream is initialized with the string given as argument to `outstring`.

  This is a souped up version of the output stream example at the beginning of the section.

• **inoutprocess** : string * string list -> ![get:int -> string,
getline:unit -> string,
put:string -> int,
flush:unit -> unit,
kill:unit -> unit]

  creates a process which will execute the shell command given in argument in parallel with the current HimML process. This shell command is given in the form (command name, arguments). The command name is searched for, using the PATH shell variable, with arguments as given.

  All text that is printed on the parallel process’s standard output can be read by the HimML process by using the `get` and `getline` methods; moreover, HimML can sent data, as text, by writing to its standard input with the `put` method, while `flush` empties the output buffers to really send the text to the process; and the latter can be terminated by calling the `kill` method.

  Contrarily to files, which can be closed and then revived when necessary, a killed process cannot be revived, and an IO exception will be raised when attempting to read from a dead process. For other remarks, see items `inprocess` and `outprocess`.

• **delete** : string -> unit deletes the file whose name is given in argument. If any I/O error occurs, then the exception IO is raised, applied to the corresponding error code. In particular, if the argument is the name of a directory, `rmdir` should be used instead.

• **rename** : string * string -> unit renames the file whose name is given as first argument to the name given as second argument. If any I/O error occurs, then the exception IO is raised, applied to the corresponding error code. Note that the capabilities of `rename` vary greatly from system to system. For example, `rename` can move files from any place to any other place on BSD Unix; this is restricted on Unix System V, Amiga and Mac systems to move files only inside the same volume (file system).

• **filetype** : string -> string set takes the name of a file and returns a set of properties of this file as character strings. If the file does not exist or any other I/O error occurs, then the exception IO is raised, applied to the corresponding error code. Otherwise, the following strings are used for properties:

  - "a" means the file is an archive (Amiga only);
  - "b" means this is a block special file (Unix only);
  - "c" means this is a character special file (Unix only);
  - "d" means the file is a directory;
  - "e" means the file is erasable, either by delete or by `rmdir` (by the owner, on Unix systems; by all, on other systems); on Unix, "eg" means the file is writable by users in the same group, "eo" means the file is writable by all other users.
  - "f" means this is a fifo, a.k.a. a pipe (Unix only);
- "g" means the file has the setgid bit set (Unix only);
- "l" means the file is a symbolic link (Unix only);
- "n" means this is a regular (normal) file;
- "p" means the file is pure (Amiga only);
- "r" means the file is readable (by the owner, on Unix systems; by all, on other systems); on Unix, "rg" means the file is readable by users in the same group, "ro" means the file is readable by all other users.
- "s" means the file is a script file (Amiga only);
- "S" means the file is a socket (Unix only);
- "u" means the file has the setuid bit set (Unix only);
- "v" means the file has the “save swapped text after use” option (Unix only, should be obsolete);
- "w" means the file is writable (by the owner, on Unix systems; by all, on other systems); on Unix systems, if a file is writable, it is also erasable; on Unix again, "wg" means the file is writable by users in the same group, "wo" means the file is writable by all other users.
- "x" means the file is executable (by the owner, on Unix systems; by all, on other systems); on Unix again, "xg" means the file is writable by users in the same group, "xo" means the file is writable by all other users.

- **dir** : string -> string set takes a path as input, which should be a correct path name for a directory, and returns the set of all file names (except . and .. on Unix systems) inside this directory. If any I/O error occurs, then the exception **IO** is raised, applied to the corresponding error code.
- **cd** : string -> unit changed the current directory to the one specified by the path given as input. If any I/O error occurs, then the exception **IO** is raised, applied to the corresponding error code.
- **pwd** : unit -> string returns a name for the current directory, as changed by **cd**. If any I/O error occurs, then the exception **IO** is raised, applied to the corresponding error code. This may notably be the case if the path name is too long.
- **mkdir** : string -> unit creates a new directory by the name given in argument. If any I/O error occurs, then the exception **IO** is raised, applied to the corresponding error code.
- **rmdir** : string -> unit creates a new directory by the name given in argument. If any I/O error occurs, then the exception **IO** is raised, applied to the corresponding error code. In particular, on most operating systems, **rmdir** can be used on empty directories only. To delete files, use **delete**.
- **system** : string -> int issues the shell command given as argument. On Unix systems, this calls the **system(3S)** function, which calls a sh-compatible shell to execute the command. **system** launches the command, waits for it to return, and returns the exit code of the command (0 if all went well). If an error occurs (not enough memory, not enough processes available, command interrupted by a signal, typically), then an **IO** exception is raised.
- **getenv** : string -> string option reads the value of the environment variable whose name is given, and returns **NONE** if it has not been defined, and **SOME** of its value, otherwise.
- **args** : unit -> string list returns the list of command-line options given after the -- switch on HimML's command-line.
- **iomsg** : int -> string translates the I/O error code in argument (as returned as argument of a **IO** exception) into a string in human-readable form. This is the same message as the one printed by the C function **perror()** on Unix systems.
• **leftmargin : int ref** defines the left margin for printing, as a number of spaces to print at the beginning of a new line when pretty-printing. It is ref 0 by default.

• **rightmargin : int ref** defines the right margin for printing, as a number of columns (counted as characters) from the beginning of the line where a new line has to be forced by the pretty-printing functions. It is ref 80 by default.

• **maxprintlines : int ref** defines the limit on the number of lines printed by pretty-printing functions, mainly to avoid looping while printing infinite structures, or to avoid printing structures of humongous sizes fully. This is also valid for printing values defined on the toplevel, since pretty is used for this purpose. The default value is ref 100. To suppress the limit in practice, write `maxprintlines := max_int`.

• **numformat : string ref** is a reference to the string that is used to format floating-point values for printing. Any C-style format for printing doubles may be used, i.e. it is \%[- | + | | # | *[0 \text{-} 9] * (\text{\textbf{-}0\text{-}9})|][feEgG], where – forces left adjustment, + forces a sign in front of the value, a blank puts a space instead of a plus sign in front of the value, a hash sign (#) forces a radix character (i.e. a dot in general; besides, trailing zeroes are not removed in the case of g and G conversions); the following optional decimal number specifies the minimum field width (with the left adjustment flag –, the number is padded on the right if it is not large enough); if this is followed by a dot, and a decimal number, this specifies the number of digits to appear after the radix character for e and f conversions, the maximum number of significant digits for the g conversion. The possible conversion characters are: f prints in the format \[ \text{\textbf{-}} ddd.ddd \] with 6 digits by default (this can be modified by precision specifications; 0 says not to output the radix character); e or E print in the format \[ d.ddd\text{\textbf{e}}+ jdd \] (or with E instead of e, respectively), with one digit before the radix character, and 6 digits by default; g or G choose between f and e (resp. E) conversions: the latter is chosen if the exponent is less than \(-4\) or greater than or equal to the precision (by default, 6). The default is "%G".

### 4.10 Miscellaneous

Interesting values are:

• **it : unit** is a special variable containing the last expression evaluated at toplevel. It is implicitly redefined every time an expression (not a declaration) is input at toplevel. Indeed, an expression e at toplevel is conceptually equivalent to writing the declaration:

\[
\text{val it = e}
\]

• **features : |

```plaintext
OS : string,
retyping : string,
continuations : string set,
numbers : string set,
structures : string set,
profiling : string set,
debugging : string set,
floatformat : string,
version : int * int * string * int,
recompiled : string,
started : string,
maintenance : string,
creator : string|
```

(may be extended in future versions) is a description of the features in the implementation and the session. Its use is informative, but it may also serve for conditional compilation. The fields currently defined are:
- **OS** defines the operating system for which this HimML compiler was compiled; it may be "Mac", "Amiga", "Unix System V" or "Unix BSD" for now. Currently, The Mac port is lagging behind, due to the demise of my old Mac II. A port for PCs should be possible, though there seems to be some major hurdles to overcome (mainly because of segmenting schemes).

- **reftyping** defines the scheme used to handle types of imperative constructs in HimML. Its value may be "imperative tyvars", meaning the Standard ML typing scheme [6], "weak tyvars", meaning the Standard ML of New Jersey typing scheme with weak type variables (assumed throughout this document), or "effect tyvars", meaning typing with more precise effect descriptions[14], or yet another, not yet implemented.

Frankly, the latter was never fully implemented, and while the weak tyvars discipline remained the default for years, and was buggy and not really required. Today, HimML uses the imperative tyvars discipline.

- **continuations** defines the style of continuation reifying mechanism. It is a set of strings, which may contain "callcc" (callcc reifies the whole functional state of evaluation, i.e. the whole stack) and "catch" (catch does the same as callcc, but the continuation it reifies is valid only while we are still evaluating the catch call).

- **numbers** is a set of strings defining the styles of numerical objects we may handle: it may contain "complex floating point", which defines complex floating point values with attached dimensions and scales, and also "big rationals" which defines infinite precision rational numbers. Only the first of these options has been implemented.

- **structures** defines the structuring mechanism used in HimML (modules). Its value is a set of strings that may contain "Standard ML structures", indicating the structure system described in [6] and [9], and "separate compilation a la CaML", indicating we have a separate compilation module system a la CaML. The first (not yet implemented) provides a sound way of structuring name spaces for types and values. The second only provides separate compilation facilities. It is described in Section 5.5.

- **profiling** defines the profiling possibilities. It is a set that may contain:
  * "call counts", meaning that the number of times each function is called is tallied,
  * "proper times", meaning that a statistical estimation of the time spent in each function (excluding any other profiled function it has called) is computed,
  * "total times" (same, but including time spent in every called function),
  * "proper allocations" and "total allocations". The last two options mean that memory consumption is recorded, as well, but have not been implemented yet. See Section 5.4 for a detailed description of the profiler.

- **debugging** defines the debugging support. It may contain:
  * "standard", which means we can set breakpoints and read values at breakpoints, and we can consult the call stack;
  * and "replaying debugger", if the debugger may reverse execution up to specified control points to show the origins of a bug. Only the last option has not been implemented in HimML. (See Section 5.3 for a description of the debugger.)

- **floatformat** defines the format of floating point numbers. This is important since certain numerical features are not provided in certain modes, and since the management of rounding may differ with a format or another. The floatformat may be "IEEE 754" or "unknown".

- **version** defines the current version of HimML. It is composed of:
  * a major version (currently 1), incremented each time a fundamental decision is made by the creator of the language which affects deeply the way we program in it (deletions or modifications of values or constructions in the language may qualify if they are important enough)
  * a minor version (currently 0), incremented for any minor revision (addition of functionalities qualify; deletion or modification of minor functions qualify too)
the code status, a string which may be:
"alpha", if only the creator (or close friends) have access to the implementation;
"beta", if released for beta-testing to a group of people;
"gamma", if deemed enough stable and bug-free to be used by final users;
"release", if distributable.

the revision (currently 11), incremented each time a bug is corrected or something is made better inside the language, without changing its overall behavior.
- recompiled is the last date of recompilation of this revision of HimML.
- started is the date the current session was started. This may be used inside a HimML program to see if it has been restarted from disk or if it has just been transferred to another session.
- maintenance is a description of the person or service the user should contact if there is any problem with HimML.
- creator is the name of the creator of the language, that is, "Jean Goubault".

- times : unit -> time * time computes the user time (first component of the returned pair) and the system time (second component). On Unix systems, this time takes its origin at the moment the current HimML process was launched. On other systems, the origin may be the time at which the machine was last booted. These times are in seconds, the default scale of dimension time, defined as:

  \text{dimension time(s)}

- force : 'a promise -> 'a forces evaluation of a promise (created with the delay construct), and returns the result. A delayed expression is evaluated at most once: the result of forcing is memoized.

- callcc : ('1a cont -> '1a) -> '1a reifies the current continuation, and passes it to the function in argument. This continuation includes a copy of the current stack of exception handlers, so that triggering a continuation reinstall the exception handlers in use when reifying the continuation: this is the reason why imperative types are used.

  The extent of the continuation is indefinite, i.e. it can be thrown at any time, even after the call to callcc is completed.

- catch : ('1a cont -> '1a) -> '1a reifies the current continuation, and passes it to the function in argument. This continuation includes a copy of the current stack of exception handlers, so that triggering a continuations reinstall the exception handlers in use when reifying the continuation: this is the reason why imperative types are used.

  Contrarily to callcc, the extent of the continuation is not indefinite, that is, throwing the reified continuation is valid only until the call to catch is completed. Afterwards, any attempt to throw the continuation may raise the exception Catch. (This is not systematic, however.)

  catch is the basic mechanism for implementing threads, a.k.a. coroutines, i.e. separate computations sharing the same process space, and which can be suspended or resumed at will. The current implementation actually builds catch as basically a thread mechanism, where the process stack is split up in several different stack zones, called thread caches, and thread switching is implemented as changing caches. callcc is implemented in the same way, except that thread caches are copied back to the heap for possible future reuse. Thread caches may also be copied to the heap to make room for a new thread when there is no remaining available cache in catch or callcc, and they are copied back from the heap when executing throw.

- throw : 'a cont -> 'a -> 'b reactivates a continuation, by passing it the return value of the continuation.
• `gc : unit -> unit` forces a garbage collection—not necessarily a major one—to take place. This is usually not needed. It may be used in alpha status to scramble the heap and see if it incurs any nasty memory management bug in the HimML run-time. It may also be used prior to doing some benchmarks, but as of yet, there is no guarantee that the garbage collection will indeed collect every free object (this does not force a major collection).

• `major_gc : string -> unit` forces a major garbage collection. This can be used in theory to get back some memory that you know can be freed right away, but it takes time. Its main use is in conjunction with the `-gctrace <file-name>` command-line option: then each garbage collection will write some statistics to `<file-name>`, which can be used to detect space leaks. Typically, when you wish to see how much memory is allocated or freed in a call to some function `f`, call `major_gc "before f"` before calling `f`, and `major_gc "after f"` afterwards. Then consult `<file-name>`. There should be some information on the status of memory before `f`, beginning with a line of the form:

```
==[before f]=========================
```

then sometime later another block of information on the status of memory after the call to `f`, beginning with a line of the form:

```
==[after f]=========================
```

• `abort : unit -> 'a` aborts the current computation, and returns to toplevel without completing the current toplevel declaration (actually, it invokes the toplevel continuation). This is the function that is internally used when syntax errors, type errors or uncaught exceptions are triggered. Following the definition of Standard ML, the current toplevel declaration is abandoned, so that its effect are nil, except possibly on the store.

• `quit : int -> 'a` aborts all computations, and returns to the operating system with the return code in argument.

• `stamp_world : unit -> unit` internally records a checksum and a copy of the current HimML environment. All modules (see Section 5.5) are compiled in the latest stamped copy of the HimML environment, and are saved to disk with the corresponding checksum. This way, when a module is reloaded, its checksum is compared to the checksums of all stamped environments in memory, and an error is triggered if no environment exists with the same checksum (meaning that we cannot safely use the module, either because it needed a richer or incompatible environment, or because the version of HimML has changed too much). The environment is automatically stamped just before HimML starts up.

• `system_less : ''a * ''a -> bool` is the system order. It returns `true` if the first argument is less than the second in the system order. You may use this ordering as a total ordering on all data of equality admitting types.

Another total ordering on equality admitting types is

```haskell
fun system_less_2 (x,y) = 
    case (x,y) of {z,...} => z=x
```

but this is slower. It is also a different order.

• `inline_limit : int ref` is a reference to a limit on the size of functions that the compiler will accept to inline. Its default value is 30. (Units are unspecified, but are roughly the numbers of basic operations and of variables in the function.) In version 1.0 alpha 20, there is no compiler yet, but one that produces C source code to feed to your favorite C compiler is in the works.
Chapter 5

Running the System

5.1 Starting up

HimML runs on Unix systems, and Amigas. Previous versions also worked on Apple Macintoshes, but this one lacks some functions. On a Mac, the only way to launch a HimML session is to double-click on the HimML icon; a text window opens, asking to enter Unix command-style arguments: enter the arguments to the HimML command, except the command’s name itself. From then on, all work happens in this console, at toplevel, as on Amiga and Unix systems.

On Amigas and Unix boxes, type `himml` followed by a list of arguments. The legal arguments are obtained by typing `himml ?`, to which HimML should answer:

```
Usage: himml [-replay replay-file] [-mem memory-size]
[-cmd ML-command-string] [-init ML-init-string] [-path path]
[-col number-of-columns]
[-grow memory-grow-factor] [-maxgrow max-memory-grow-factor]
[-nthreads max-cached-threads] [-threadsize thread-cache-size]
[-maxcells max-cells] [?] [-gctrace file-name]
[-pair-hash-size #entries] [-int-hash-size #entries]
[-real-hash-size #entries]
[-string-hash-size #entries] [-array-hash-size #entries]
[-pwd-prompt format-string] [-core-trace] [-data-hash-size #entries]
[-c source-file-name] [-inline-limit max-inlined-size] [-nodebug]
[-- arguments...]
```

and exit. Launching HimML without any arguments is fine. There are other HimML tools, used to compile, link and execute bytecode compiled files; they are listed at the end of this section.

To load a file, the `use` keyword may be used; it begins a declaration, just like `val` or `type`, that asks HimML to load a file and interpret it as if it were input at the keyboard (except it does not use `stdin`. The path that `use` uses can be extended on the command line by the `-path` switch, or inside HimML by changing the contents of `usepath : string list ref`, which is a reference to the list of volumes or directories in which to search for files, from left to right.

The explanation of the various options are:

- `--replay <replay-file>`: whenever himml is executed, it records every single word it parses in a `replay file`, named HimML.trace by default; this is useful notably in debugging the language (see section 5.9), but also to see a computation evolve (because HimML writes there its garbage collection statistics as ML comments). To replay such a file, the `--replay` option is used with the name of the replay file. This option is incompatible with all the others.
-core-trace instructs HimML to create and fill in a replay file for use with the -replay option. The replay file is named HimML.trace and is created in the current directory. It records every single line typed on standard input or in any included file (by use).

-mem <memory-size> sets the initial amount of memory HimML takes for its heap. By default, it is 400000 (400 Kbytes) on Unix systems, and this is expanded on demand by chunks of 400 Kbytes. On Macs, where the heap cannot be expanded, the initial memory size is the maximum memory size; by default, it is 16000000 (16 Mbytes), and the system tries to allocate a heap at most that large on launching. If it cannot do it, it reduces automatically the figure until the heap can be allocated or until it finds out there is not enough memory to load. On Amigas, the policy is like on Unix systems.

-mem can be used only once on the command line.

-maxcells <max-cells> sets an upper bound on the number of cells to allocate. This limit is not strict: if the system feels it absolutely needs more memory, it will grab some, but only a minimal amount of it, to avoid aborting the current HimML process. By default, the number of cells is unlimited, but it may be useful in some situations to limit it, otherwise HimML will take as much memory as it wants to feel at ease, without considering any other processes on the same machine.

-maxcells can be used only once on the command line.

-nthreads <max-cached-threads> sets the number of cache threads that are available for the implementation of callcc and catch. In the current implementation, the process stack is split into several distinct stack regions that act as stacks for threads, and callcc and catch use these as caches for heap-based thread stacks (the more, the better). By default, there are 10 such caches, i.e. the default option is -nthreads 10.

-nthreads can be used only once on the command line.

-threadsize <thread-cache-size> sets the size in bytes of the cache threads that are available for the implementation of callcc and catch. In the current implementation, the process stack is split into several distinct stack regions that act as stacks for threads, and callcc and catch use these as caches for heap-based thread stacks. The larger the thread caches, the less often cache overflows will occur (in which case, catch is used transparently to switch threads and resume computation in a new empty thread); the smaller the thread caches, the faster catch and mostly callcc will be (the former might have to copy thread caches to or from memory; the latter needs to). By default, thread caches are 20000 bytes wide, with 4000 bytes left as safety zone; i.e., the default option is -threadsize 20000.

-threadsize can be used only once on the command line.

-pair-hash-size <#entries> sets the number of slots in the global hash-table that is used to keep a record of all shared pairs (pairs, list cells, basic blocks for sets, and so on). You may wish to give it a value higher than the default (typically 23227) for memory-hungry programs. A rule of thumb is to evaluate how many cells your program needs (one tuple is one cell, a n-element list uses up n cells, a n-element set or map uses up some 2n cells; or, more practically, run HimML with the -gctrace option on, and look at the statistics on total live homo cells), and then to divide this number by, say, 3.

On the other hand, having a big table for few values makes for longer garbage collection times, so you may also wish to reduce this value on programs that do not use much memory, or which only allocate very short-lived data.

-int-hash-size <#entries> is the same as -pair-hash-size, except it is concerned with integers. (All integers are boxed and shared in HimML.)

You may wish to raise its value if your program builds and keeps lots of integers in data-structures, or you may wish to decrease its value if you use few integers or only allocate them for temporary computations.

-real-hash-size <#entries> is the same as -pair-hash-size, except it is concerned with reals (floating-point values).
You may wish to raise its value if your program builds and keeps lots of numbers in data-structures, or you may wish to decrease its value if you use few numerical quantities or only allocate them for temporary computations. (In particular, there is no need to increase its value for ordinary number-crunching, except if you are handling big matrices.)

- `string-hash-size <#entries>` is the same as `pair-hash-size`, except it is concerned with strings. You may wish to raise its value if your program builds and keeps lots of strings, or does a lot of text processing. It is not advised to reduce its value, as many strings are used internally by the compiler and the type-checker.

- `array-hash-size <#entries>` is the same as `pair-hash-size`, except it is concerned with arrays (in general, with n-tuples, n ≥ 3, or records with at least two fields, or arrays with at least 3 entries). You may wish to raise its value if your program builds and keeps lots of tuples, records and arrays, or you may wish to decrease its value in case you don’t use many of these structures.

- `cmd <ML-command-string>` is used to launch HimML as a batch process. It makes HimML execute the program whose text appears in the string `<ML-command-string>`, and exit upon termination. The string is parsed and executed as if it were input at the keyboard; e.g., this might be of the form "use "myfile.ml";", where ‘myfile.ml’ contains declarations and a call to the main function in the project. No welcome banner, no result of typing or evaluation and no spurious message is printed; to print a message, you must use the input/output functions. Moreover, standard input is not used by the parser, and can be read by input/output functions.

- `cmd` can be used only once on the command line, and is incompatible with `init`.

- `init <ML-init-string>` is used to initialize a HimML process. It makes HimML execute the program whose text appears in the string `<ML-command-string>`, and then present the usual toplevel interface. The string is parsed and executed as if it were input at the keyboard, though standard input is not used to this end; typically, this string will be of the form "use "myfile.ml";", where ‘myfile.ml’ contains declarations.

- `init` can be used only once on the command line, and is incompatible with `cmd`.

- `path <path>` instructs HimML to look for files to load (by the `use` keyword) in the directory `<path>` if it didn’t find them before. The current directory is always the first searched directory. Then come the paths specified on the command line, in the order in which they arrive.

- `gctrace <file-name>` is an option that is off by default. If you specify it, this turns it on: then, each time HimML will trigger a garbage collection, some information will be written to the specified file. This information is a sequence of lines of the form:

```
========================
GC...done:
  total number of allocated memory cells [nCells] = 54272
  allocated homo cells in young generation : 4608 (~73728 bytes, not counting sharing overhead)
  allocated hetero cells in young generation : 512 (~8192 bytes, not counting sharing and contents overhead)
  live homo cells in young generation : 4019
  live hetero cells in young generation : 25
  total live homo cells : 4019
  total live hetero cells : 25
  strings freed : 43 bytes
  patcheckbits freed : 0 bytes
  stacks freed : 360 bytes
  vectors (environments, arrays, tuples, records) freed : 6608 bytes
```
The latter means the following: one garbage collection has just been done (if the system crashes during a GC, you will just get GC...), the number of cells in the system is 54272, of which 4608 + 512 = 5120 are considered young (i.e., will be considered as highly likely to become garbageable at the next GC); among these, 4019 + 25 = 4044 are live, i.e., not free. And the system as a whole also contains 4019 + 25 = 4044 live cells. The purpose of the “homo” and “hetero” figures is to separate between homogeneous cells (pairs, integers, maps, reals, complexes, etc.) and heterogeneous cells (which point to non-first class data, like strings, which point to an area of memory where its contents lies, or arrays, or $n$-tuples with $n \geq 3$, or records with at least 2 fields, which are allocated as a cell pointing to an internal array of values). For hetero cells, the amount of additional memory freed is shown: 43 bytes of strings, none of patchcheckbits (an internal structure of the compiler), 360 bytes of stacks (i.e., of local thread structure), 6608 bytes of vectors, and 16 externals were freed. Externals are interfaces between HimML and non-HimML data, typically files. The time taken to do this garbage collection was 0.089s., and the heap had only one generation (the so-called young generation) before garbage collection, and is segmented in two generations afterwards. It allocated 7 local threads since startup (it allocates at least one at each toplevel command), most of which have been freed since then. And it allocated and freed so many bytes of temporary storage (typically for local HimML variables during execution of code), of which 52400 remain allocated at the end of garbage collection.

Calling major_gc invokes a major collection, and the argument passed to major_gc is printed at the start of the information block, e.g.:

```bash
==[test]========================= GC...done:
  total number of allocated memory cells [nCells] = 54272
  allocated homo cells in young generation : 36864 (~589824 bytes, not counting sharing overhead)
  allocated hetero cells in young generation : 512 (~8192 bytes, not counting sharing and contents overhead)
  live homo cells in young generation : 6145
  live hetero cells in young generation : 14
  total live homo cells : 6145
  total live hetero cells : 14
  strings freed : 29 bytes
  patchcheckbits freed : 0 bytes
  stacks freed : 240 bytes
  vectors (environments, arrays, tuples, records) freed : 156204 bytes
  16 externals freed
  garbage collection time = 0.119s.

There were 1 old generations, plus one new;
there are 1 old generations, plus one new.
Number of stacks (threads) allocated since startup: 11
Number of allocated bytes of temporary (stack) storage: 118832
```

- `-col <number-of-columns>` specifies the width of the screen of a HimML session, in characters (by default, 80). This is used by the HimML toplevel when it prints types and values, and by the debugger.
- `-grow <memory-grow-factor>` specifies the initial ratio of the size of the heap to the size occupied by live data that the garbage collector tries to maintain. By default, it is 2.0. The greater the number, the less time will
be spent in garbage collection overall, but the more time a single garbage collection may take. This number can not go lower than 1.0, and evolves across garbage collections to adapt to the evolving nature of the computations.

- **-maxgrow<max-memory-grow-factor>** puts an upper bound to the ratio of the size of the heap to the size of the live data space. By default, it is 8.0.

- **-pwd-prompt** followed by a format string tells HimML that it should use a prompt that mentions the HimML current directory (as modified by the HimML cd function and read by the HimML pwd function). The HimML Emacs mode uses a format string starting with the escape character and continuing with |%s|%s: the first %s will be replaced by the HimML current directory, the second by the current prompt (normally,>). This is used by Emacs to synchronize its current directory with HimML’s in HimML mode.

- **-inline-limit<limit>** installs a new size limit that the compiler reads when deciding whether it should inline functions or not. This is essentially the same as setting inline_limit to <limit> at HimML initialization time.

- **-c<himml-source-file>** compiles the given source file, and produces a compiled module file: see Section 5.5.3.

- **-nodebug** disables the debugger: typing control-C will still interrupt the currently running HimML program, but instead of entering the debugger, it will stop the program. Moreover, raised exceptions won’t enter the debugger either.

Finally, using -nodebug will direct the bytecode compiler not to output any debugging information. This can be used to produce stripped modules (i.e., without any debugging information), typically to save space or to prevent or make reverse engineering of production code difficult.

Note that, if you compile a module with -nodebug, and execute it under HimML (with the debugger on), then typing control-C or raising non-benign exceptions will enter the debugger, but the debugger won’t be able to extract any information from the compiled code.

- **--** stops parsing of all options, and instructs HimML that the rest of the command-line consists of options and arguments that will be available from HimML programs by looking at the list args().

### 5.2 Compiling, Linking, Finding Dependencies

As said earlier, the HimML distribution includes other tools to compile, link and run bytecode compiled files:

- **himml -c** compiles a module (see Section 5.5 for details). That is,

  ```
  himml -c foo.ml
  ```

compiles "foo.ml", and produces a bytecode file "foo.mlx". This does exactly the same thing as typing #compile "foo.ml" at the HimML prompt; typing open "foo.ml" does almost the same thing, except HimML will then print a list of all types and identifiers defined in "foo.ml", and will declare them in the current toplevel.

- **himmlnk** links a series of bytecode files into one; this works both as a linker and as an archiver. Syntax is:

  ```
  himmlnk archive-file file1.mlx ... fileN.mlx
  ```

to create an archive file—in which case it is recommended to give it a .mla extension—, or a bytecode executable file.
• **himmlrun** runs the HimML bytecode interpreter: `himmlrun "foo.mlx"` followed by arguments will execute the `main ()` function in file "foo.mlx" (that the name ends in .mlx is, by the way, totally irrelevant), and the HimML `args()` function will get back the command-line arguments. There is in fact no need to explicitly call `himmlrun`, as (at least on Unix) launching "foo.mlx" will invoke `himmlrun` automatically, if properly installed.

• **himmldep** computes dependencies between HimML source files. This is used in building makefiles, as used by `make`.

  A typical use of himmldep is to run

  ```
  himmldep * .ml > .depend
  ```

  at the (Unix) command-line. This will produce a file .depend listing all dependencies between files, which can be used by `make` to help reconstruct all proper .mlx files.

  In fact, the standard makefile for projects using HimML is as follows:

  ```
  %.mlx : %.ml
      himml -c $<
  
  OBJS = a.ml b.ml c.ml
  
  prog: $(OBJS)
      himmlnk prog $(OBJS)
  
  clean:
      -rm *.mlx prog
  
  cleanall : clean
      -rm *
  
  depend:
      himmldep *.ml > .depend
  
  include .depend
  ```

  The first line (works only with GNU make) tells make that to build or rebuild any bytecode file, say foo.mlx, it should call `himml -c foo.ml`. The OBJS = line is a macro definition, stating what bytecode files we would like to build. The prog: line states the main rule, which is to build a HimML executable file or a library file prog, by calling himmlnk to link all bytecode files in OBJS. The clean and cleanall are targets meant to remove compiled files, and are called with `make clean` or `make cleanall` respectively. Dependencies are recomputed by typing `make depend`, which creates dependencies in the .depend file; the latter is in turn included in the current makefile using GNU make's include directive.

  If you don't have GNU make, then you cannot include .depend, and you will have to copy its contents manually at the end of makefile. Additionally, the %.mlx : %.ml line should be replaced by:

  ```
  .SUFFIXES: .ml .mlx
  
  .mlx.ml:
      himml -c $<
  ```
5.3 Debugger

HimML contains a debugger, as shown by consulting the set #debugging features, which should be non-empty. It can be called by the break function:

- break : unit -> unit enters the debugger.

Another way of entering the debugger is when an exception is raised but not caught by any handler.

There are two ways of entering the debugger. These are shown on entry by a message, stop on break (we entered the debugger through break, or by typing control-C or DEL when evaluating an expression), or stop at... (we entered the debugger at a breakpoint located just before the execution of an expression).

In any case, the debugger enters a command loop, under which you can examine the values of expressions, see the call stack, step through code, set breakpoints, resume or abort execution. The debugger presents a prompt, normally (debug). It then waits for a line to be typed, followed by a carriage return, and executes the corresponding command. These commands are:

- h, or help, displays a summary of all debugger commands. This is no replacement of the general documentation (this document), its aim is to remind the user of the particular commands the debugger offers. The summary is also displayed whenever the debugger did not understand your last command.

- s, or step, resumes execution in single-step mode. That is, the current execution is resumed, and after executing a single instruction, the debugger is entered again. When coming upon a procedure call, s enters the procedure and stops on entry.

- n, or next, resumes execution in single-step mode. That is, the current execution is resumed, and after executing a single instruction, the debugger is entered again. When coming upon a procedure call, n evaluates the call without entering the procedure, and stops at the beginning of the next instruction. When the called procedure raises an exception that is not caught inside the procedure, the debugger manages to stop at the beginning of the next instruction to be executed, which is not necessarily the one just after the breakpoint—it will probably be one at the beginning of an exception handler.

- c, or cont, or continue resumes execution, as though the debugger had not been called. In particular, it does not single-step through the code. If the debugger was entered through break, execution is resumed as though we had never entered the debugger, and break acts as the identity function. If the debugger was entered through a breakpoint, execution is resumed. If it was entered through an uncaught exception, execution is aborted, and control returns to the toplevel.

The c command may take an argument, which should be a HimML expression e. This expression is parsed, type-checked, compiled and evaluated in the current environment (which is the environment as seen from the point where execution was stopped; but see the u, d and w commands). No breakpoint in the expression is ever triggered, and interrupting its evaluation by control-C or DEL just cancels the evaluation and returns to the debugger, without, say, entering a recursive level of debugging.

If the expression successfully evaluates, the resulting value v is returned as the result of the expression on which execution was stopped. This means the following: on a stop on break, the return value is replaced by v, and execution is resumed with this new value instead of the previously computed one; on a stop on entry to an expression e', the expression e' is not evaluated, and v masquerades as the value that e' should have (this is useful when e' is not a reliable piece of code, but we know in advance what it should return and we don’t want to lose time debugging e').

Note that, although there is some type-checking involved in the evaluation of the expression e, this only provides a relative, not absolute, level of safety. That is, type-checking under the debugger may catch some type errors, but not all (in short, the debugger is not type-safe). For example:

```haskell
exception Div0
fun inv x = if x=0.0 then raise Div0 else 1.0/x
inv 0.0
```
will enter the debugger at the raise expression, and \( e \) will be coerced to the finest type the debugger can infer from the definition of \( \text{inv} \) alone, that is, \( \#a \rightarrow \#a' \sim 1 \) (since \( \text{inv} : \#a \rightarrow \#a' \sim 1 \)). However, the only allowable type in the current context would be \( \text{num} \). So if you type \( c \ 35 \text{cm} \), \( \text{inv} \) will return with a mostly unpredictatable value. Even more seriously, in other cases this may cause the HimML system to crash, although this risk is limited because there are run-time safeguards against that in the HimML system when the debugger is present.

- **p**, or print, prints the value of the HimML expression that follows. This is useful notably to display values of variables or of components of values of variables. However, any HimML expression, including expressions with side-effects, or that print values in a special format and return \( () \) afterwards, can be given as argument to \( p \): this allows to actually execute any expression from inside the debugger.

  The argument expression is parsed, type-checked, compiled and evaluated in the current environment (which is the environment as seen from the point where execution was stopped; but see the \( u \), \( d \) and \( w \) commands). No breakpoint in the expression is ever triggered, and interrupting its evaluation by control-C or DEL just cancels the evaluation and returns to the debugger, without, say, entering a recursive level of debugging.

  As for the \( c \) command, the type-checker can only provide a relative, not absolute, level of type-safety, and it is possible to evaluate non-sensical expressions because the type-checker cannot hope to detect all possible type errors. This is because the debugger uses only the types that have been inferred statically, but it cannot specialize them to the real run-time types.

- **b**, or break, sets a breakpoint on entry to an expression. This expression is referenced by function name (immediately following a \( @ \) character), a line number and a column number (immediately following a \( : \) character), all of which being optional. If the function name is omitted, the current function is assumed (as shown at level 1 of the call stack by the \( w \) command, or as specified by the last function listed by the \( l \) command). If the line number is omitted, the first line is assumed; if the column number is omitted, the breakpoint is set as far left on the line as possible.

  After successfully setting the breakpoint, it is associated a breakpoint number, which is then shown between square brackets, followed by information about the breakpoint location. If no breakpoint could be set at the indicated location, the debugger won’t install any breakpoint, and will say so.

  - **sh** or showbreakpoints, shows all currently set breakpoints.
  - **e**, or erase, erases the breakpoint whose number is given as argument.
  - **w**, or where, shows the call stack at the point where execution was stopped. The levels inside the call stack are numbered from 1 (current scope, shown first; we say it’s the bottom of the stack, although it is shown first, and therefore on the top of the other levels) to toplevel.

    The commands \( u \) and \( d \) move up and down the stack respectively, and the current level is shown in the stack display by being prepended the \( > \) character. This shows the environment that will be used to type-check, compile and evaluate expressions, as with the \( p \) print command or the \( c \) continue command. By default, the current level is 1 (bottom of stack).

    By default, \( w \) only shows 10 levels of stack. This is to avoid huge stack dumps in the case of infinite recursions. You can specify another depth limit by giving \( w \) a numeric argument.

  - **u**, or up, moves up the specified number of levels (1 by default) in the call stack: see \( w \).
  - **d**, or down, moves down the specified number of levels (1 by default) in the call stack: see \( w \).
  - **l**, or list, lists the function whose name is given as argument, with line numbers in front of each line, so as to help set breakpoints or spot where the debugger was stopped. If there are several functions with the same names in the various environments lying in the call stack, the one that is in scope relative to the current stack level is listed.

    This also sets the current function to the specified one, so that the \( b \) command can then be issued to set a breakpoint in this function without having to retype the name of the function.
- `q` or `quit` leaves the debugger and aborts the current computation, returning to toplevel. This does not quit the HimML session, as the `quit` function, it just leaves the debugger and returns to the toplevel.

The way that the interpreter gives control to the debugger is by means of code points, which are points in the code where the compiler adds extra instructions. These instructions usually do nothing. When you set a breakpoint, they are patched to become the equivalent of `break`. Alternatively, these instructions also enter the debugger when we are single-stepping through some code.

These instructions are added by default by the compiler, but they tend to slow the interpreter. If you wish to dispense with debugging information, you may issue the directive:

```
(*$D-*)
```

which turns off generation of debugging information (of code points). If you wish to reinclude debugging information, type:

```
(*$D++)
```

These directives are seen as declarations by the compiler, just like `val` or `type` declarations. As such, they obey the same scope rules. It is recommended to use them in a properly scoped fashion, either inside a `let` or `local` expression, or confined in a module.

### 5.4 Profiler

The way that the interpreter records profiling information is by means of special instructions that do the tallying.

These instructions are not added by default by the compiler, since they tend to slow the interpreter by roughly a factor of 2, and you may not wish to gather profiling information of every piece of code you write. To use the profiler, you first have to issue the directive:

```
(*$P+)
```

which turns generation of profiling instructions on. The functions that will be profiled are exactly those that were declared with the `fun` or the `memofun` keyword.

If you wish to turn it off again, type:

```
(*$P-*)
```

These directives are seen as declarations by the compiler, just like `val` or `type` declarations. As such, they obey the same scope rules. It is recommended to use them in a properly scoped fashion, either inside a `let` or `local` expression, or confined in a module. Usually, you will want to profile a collection of modules. It is then advised to add `(*P++)` at the beginning of each. Time spent in non-profiled functions will be taken into account as though it had been spent in their profiled callers.

Then, the HimML system provides the following functions to help manage profiling data:

- `report_profiles : unit` returns the set of all profiling data that the interpreter has accumulated until now on all profiled functions. This is a dump of all internal profiling structures of the interpreter.
The **location** field describes where the function that is profiled is located. Its first component is the function name, its second component is the file name where this function was defined (or the empty string "" if this function was defined at the toplevel prompt), its third and fourth components are respectively the starting and ending positions of the definition in this file, as line/column pairs. Note that the function name alone is not enough to denote accurately which function is intended, as you can build anonymous functions (by `fn`, for example): it was chosen to let these functions inherit the name of the function in which they are textually enclosed. The file name and positions in the file are then intended to give a more precise description of what function it is that is described.

The **ncalls** shows how many times this function was called.

The **proper** and **total** fields contain statistics in the same format: time is the time spent in the function (in the format returned by `times`, i.e. user time and system time), ngcs is the number of garbage collections that were done while executing the function (this gives a rough idea of the memory consumption of the function), and gctime is the time spent garbage collecting in this function. While the statistics in **proper** only include information of what happened when the interpreter was really executing the function, **total** also includes the times spent executing all its callees.

**report_profiles** only reports statistics for those functions that were called at least once (or at least once since the last call to `reset_profiles`).

**report_profiles** is pretty low-level, and is intended to be used as a basic block for more useful report generators. One such generator is located in "Utils/profile.ml". To get a meaningful report, execute your program, then type:

```ml
open "Utils/profile";
prof stdout;
```

to get a report on your console, or:

```ml
fprof "prof.out"
```

to get a report in a file named "prof.out" in the current directory. (To open the module "profile" on a Macintosh, write `open "Utils:profile"`; in general, it’s better to modify the path to include the `Utils` directory, and not bother with directory names.)

- **reset_profiles** : unit -> unit resets all profile information, so that a new profiling round can be launched on a clear basis.

- **clear_profiles** : unit -> unit purges the system from all profiling instructions. I.e., executing the same code again won’t generate any profiling information; the code should go a bit faster, but not as fast as if it had been compiled without profiling first (it patches the profiling instructions to become no-ops).

What can you do with profile information? The main goal is to detect what takes up too much time in your code, so as to focus your efforts of optimization on what really needs it. A good strategy to do this is the following:

- Identify the functions in which the most (proper) time is spent, and optimize them.

- If the latter are already optimized, or do almost nothing, then look at the number of times they are called. Usually, such functions take time just because they are called often; then, identify their callers and rewrite them so that they don’t go through this subroutine over and over again (i.e., take shortcuts in common situations).

- Finally, in rare occasions, strange cases may occur: it may be the case that some function appears to be more costly than another one, which does the same amount of work or more. In general, this is because the interpreter needed to do some extra work behind the scenes. Typically, because it keeps on getting a full stack when entering this function, and has to switch threads (which is fast, but takes some time when done repetitively); in this case, try to make your programs less recursive—but this is really a misfeature on HimML’s part.
5.5 Separate Compilation and Modules

5.5.1 Overview

The main goal of the HimML module system is to implement separate compilation, where you can build your program as a collection of modules that you can compile independently from each other, and then link them together.

The HimML module system was designed so that it integrated well with the rest of the core language, while remaining simple and intuitive. At the time being, the HimML module system does not provide the other feature that modules are useful for, namely management of name spaces. The module system of Standard ML seems best for this purpose, although it is much more complex than the HimML module system.

Consider the following example. Assume that your program consists naturally of three files, a.ml, b.ml and c.ml. The most natural way of compiling it would be to type:

```
use "a.ml";
use "b.ml";
use "c.ml";
```

But, b.ml will probably use some types and values that were defined in a.ml, and similarly c.ml will probably use some types and values defined in a.ml or b.ml. In particular, if you want to modify a definition in a.ml, you will have to reload b.ml and c.ml to be sure that everything has been updated.

This is not dramatic when you have a few files, and provided they are not too long. But if they are long or many, this will take a lot of time. Separate compilation is the cure: with it, you can compile a.ml, b.ml, and c.ml separately, without having to reload other files first.

The paradigm that has been implemented in HimML is close to that used in CaML, and even closer in spirit to the C language. In particular, modules are just source files, as in C. Two new keywords are added to HimML: extern and open. Note that the Standard ML module system also has an open keyword, but there is no ambiguity as it is followed by a structure identifier like Foo in Standard ML, and by a module name like "foo" in HimML.

The extern keyword specifies some type or some value that we need to compile the current file, telling the type-checker and compiler that it is defined in some other file. Otherwise, if you say, for example, `val y=x+1` in b.ml, but that x is defined in a.ml, the type-checker would complain that x is undefined when compiling b.ml. To alleviate this, just precede the declaration for y by:

```
extern val x : int
```

This tells the compiler that x has to be defined in some other file, and that it will know its values only when linking all files together. This is called importing the value of x from another module.

Not only values, but datatypes can be imported:

```
extern datatype foo
```

imports a datatype foo. The compiler will then know that some other module defines a datatype (or an abstype) of this name. However, it won’t know whether this datatype admits equality, i.e. whether you can compare objects of this datatype by =. If you wish to import foo as an equality-admitting datatype, then you should write:

```
extern eqtype foo
```

Of course, if foo is a parameterized datatype, you have to declare it with its arity, for example:

```
extern datatype 'a foo
```

for a unary (not necessarily equality-preserving) datatype, or

```
extern eqtype ('a, 'b) foo
```

for an equality-preserving datatype with two type parameters.

Finally, dimensions can be imported as well:
extern dimension foo

imports foo as a dimension (type of a physical quantity, typically).

Given this, what does the following mean? We write a file "foo.ml", containing:

extern val x:int;
val y = x+1;

Then this defines a module that expects to import a value named x, of type int (alternatively, to take x as input), and will then define a new value y as x+1 and export it.

Try the following at the toplevel (be sure to place file "foo.ml" above somewhere on the load path, as referenced by the variable usepath):

val x = 4;
open "foo";

You should then see something like:

x : int
y : int
x = 4
y = 5

Opening "foo" by the open declaration above proceeded along the following steps:

- First, open precompiled the textual description of the module "foo.ml" into an object module "foo.mlx" in the same directory as "foo.ml". This object module contains, in a binary format, all information that was in "foo.ml", plus the types it has computed, as well as a representation of the interpreted code for what's in "foo.ml".

  In fact, open will recompile .mlx files from the corresponding .ml files whenever one of the .ml files on which it depends has been updated, so as to maintain consistency between the textual versions of the modules (in .ml files, usually) and their precompiled versions (the .mlx files). On the other hand, if an up-to-date .mlx file is present, it won't recompile it, and will proceed directly to the next step.

- Then, open opened the module by loading the contents of foo.mlx.

- Finally, open linked the module by resolving all extern declarations. In this example, open checked that there was a variable named x in the environment in which we issued the open declaration, checked that its type was int (to be more precise, that its type could be instantiated to int), and has defined the value x of inside the module as being the same as the value of x in the outside environment.

A variant on open is open*, which does just the same, except it does not try to recompile the source file "foo.ml": it just assumes that "foo.mlx" is up to date, or fails. This is useful when shipping compiled bytecode modules, and is used internally in the himmlpack and himmllnk tools.

Assume now that we didn't have any value x handy; then open would still have precompiled and opened the resulting object module "foo.mlx". Only, it would have failed to link it to the rest of the system. If you wish to just compile "foo.ml" without loading it and linking it, issue the directive:

#compile "foo.ml"

at the toplevel. (The # sign must be at the start of the line.) This compiles, or re-compiles, "foo.ml" and writes the result to "foo.mlx".
5.5.2 Header Files

Another problem pertaining to separate compilation is how to share information between separate modules. For example, you might want to define again three modules a.ml, b.ml and c.ml, where a.ml would define some value \( f \) (say, a function from \text{string} \text{ to } \text{int} ), and b.ml and c.ml would use it.

A first way to do this would be to write:

- in a.ml:
  
  ```
  fun f name = ... 
  ```

- in b.ml:

  ```
  extern val f : string -> int;
  ...
  f "abc" ...
  ```

- and in c.ml:

  ```
  extern val f : string -> int;
  ...
  f "foo" ...
  ```

but this approach suffers from several defects. First, no check is done that the type of \( f \) is the same in all three files; in fact, the check will eventually be performed at link time, that is, when doing:

```
open "a";
open "b";
open "c";
```

but we had rather be warned when first precompiling the modules.

Then, whenever the type of \( f \) changes in a.ml, we would have to change the \text{extern} declarations in all other files, which can be tedious and error-prone.

The idea is then to do as in the C language, namely to use one \text{header file} common to all three modules. (This approach still has one defect, and we shall see later one how we should really do.) That is, we would define an auxiliary file "a_h.ml" (although the name is not meaningful, the convention in HimML is to add \_h to a module name to get the name of a corresponding header file), which would contain only \text{extern} declarations. This file, which contains in our case:

```
extern val f : string -> int;
``` 

is then called a header file.

We then write the files above as:

- a.ml:

  ```
  use "a_h.ml"
  ```

  ```
  fun f name = ...
  ```

- in b.ml:
use "a_h.ml";

... f "abc" ...

- and in c.ml:

  use "a_h.ml";

  ... f "foo" ...

This way, there is only one place where we have to change the type of f in case we wish to do it: the header file a_h.ml.

What is the meaning of using a_h.ml in a.ml, then? Well, this is the way that type checks are effected across modules. The meaning of extern then changes: in a.ml f is defined after having been declared extern in a_h.ml, so that f is understood by HimML not as being imported, rather as being exported to other modules. This allows HimML to type-check the definition of f against its extern declaration, and at the same time to resolve the imported symbol f as the definition in a.ml. This is more or less the way it is done in C.

On thing that still does not work with this scheme, however, is how we can share datatypes. This is because datatype declarations are generative. Try the following. In a_h.ml, declare a new datatype:

datatype foo = FOO of int;
extern val x:foo;

In a.ml, define the datatype and the value x:

use "a_h.ml";

val x = FOO 3;

Now in b.ml, write:

use "a_h.ml";

val y = x : foo;

Then, open "a", then "b". This does not work: why? The reason is that the definition of the datatype foo in a_h.ml is read twice, once when compiling a.ml, then when compiling b.ml, and that both definitions created fresh datatypes (which just happen to have the same name foo). These datatypes are distinct, hence in val y = x : foo, x has the old foo type, whereas the cast to foo is to the new foo type.

The remedy is to avoid using header files, and to rather open them. So write the following in a.ml:

open "a_h";

val x = FOO 3;

and in b.ml:

open "a_h";

val y = x : foo;
Opening a_h produces a compiled module a_h.mlx, which holds the definition for foo and the declaration for x. In the compiled module, the datatype declaration for foo is precompiled, so that opening a_h does not re-generate a new datatype foo each time a_h is opened, rather it re-imports the same.

Technically, imagine that fresh datatypes are produced by pairing their name foo with a counter, so that each time we type datatype foo = FOO of int at the toplevel, we generate a type (foo,1), then (foo,2), and so on. This process is slightly changed when compiling modules, and the datatype name is paired with the name of the module instead, say, (foo,a_h). Opening a_h twice then reimports the same datatype.

The same works for exceptions, except there is no extern exception declaration. The reason is just that it would do exactly the same as what exception already does in a module. If you declare:

```haskell
exception Bar of string;
```

in a_h.ml, and import a_h as above, by writing open "a_h" in a.ml and b.ml, then both a.ml and b.ml will be able to share the exception Bar. Typing the following in a_h.ml would not work satisfactorily, since Bar would not be recognized as a constructor in patterns:

```haskell
extern val Bar : string -> exn;
```

That is, it would then become impossible to write expressions such as:

```haskell
f(x) handle Bar message => #put stdout message
```

in a.ml. However, if you don’t plan to use pattern matching on Bar, then the latter declaration is perfectly all right.

### 5.5.3 Summary

The following commands are available in HimML:

- **open** opens a module. The declaration open "foo" imports and links the compiled module foo.mlx; if the latter does not exist or is older than foo.ml, then open first recompiles the latter, producing foo.mlx, the imports and links the latter. (The name foo can of course be replaced by any other name.)

  The open keyword can also be used in local declarations, e.g.:

  ```haskell
  let val y = 3
  open "foo"
  in
  x+1
  end
  ``

  is allowed, and links the module locally. That is, assuming that foo.mlx imports y and exports x = 2*y, then the above would return 2 * 3 + 1, namely 7.

- **#compile** compiles a module. The declaration #compile "foo" compiles foo.ml, producing foo.mlx. The sharp sign # should be at the start of the line. The differences with open are that, first, the module is re-compiled unconditionally, and second, it is not attempted to load or link the resulting module.

  It is easier to compile modules by typing the following under the shell:

  ```sh
  himml -c foo.ml
  ``

  which does exactly the same as launching HimML, and typing #compile "foo"; quit 0; under the HimML toplevel.

  You can then use himml as a HimML standalone compiler, and compile each of your modules with himml -c. This is especially useful when using the make utility. A typical makefile would then look like:
.mlx : %.ml
    himml -c $<

a_h.mlx: a_h.ml
a.ml: a.ml a_h.mlx
b.ml: b.ml a_h.mlx
pack.ml: pack.ml a.ml b.ml

The first lines define a rule how to make compiled HimML modules from source files ending in .ml. It has a syntax specific to GNU make. If your make utility does not support it, replace it by:

.SUFFIXES: .mlx .ml
.ml: ml:               himml -c $<

The last lines of the above makefile represent dependencies: that a.ml depends on a.ml and a_h.mlx means that make should rebuild a.ml (from a.ml, then) whenever it is older than a.ml or a_h.mlx. Such dependencies can be found automatically by the himmldep utility. For example, the dependency line for a.ml was obtained by typing:

himmldep a.ml

at the shell prompt.

There is no specific way to link compiled modules together, since open already does a link phase. To link a.ml and b.ml, write a new module, say pack.ml, containing:

open "a";
open "b";

then compile pack.ml. The resulting pack.ml file can also be executed, provided it has no pending imported identifiers, either by launching HimML, opening pack, and running main (); (provided pack.ml exports one such function), but it is easier to type the following from the shell:

himmlrun pack

Under Unix, every module starts with the line:

#!/usr/local/bin/himmlrun

assuming that /usr/local/bin is the directory where himmlrun was installed, so that you can even make pack.ml have an executable status:

chmod a+x pack.ml

and then run it as though it were a proper executable file:

pack.ml

This will launch himmlrun on module pack.ml, find a function main and run it.

5.6 Editor Support

Any ASCII text editor can be used to write HimML sources. But an editor can also be used as an environment for HimML. In GNU Emacs, there is a special mode for Standard ML, called ‘sml-mode.el’ and that comes with the Standard ML of New Jersey distribution, that can be adapted to deal with HimML: this is the ‘ml-mode.el’ file. However, it was felt that it did not indent properly in all cases, because of the complicated nature of the ML syntax. A replacement version is in the works, called ‘himml-mode.el’; it is not yet operational.
5.7 Bugs

Remember: a feature is nothing but a documented bug! You may therefore consider the following as features :-).

- Continuations always capture the toplevel, but due to all sorts of trickeries that can be played internally, it is unsafe to capture and store continuations in the toplevel environment, and then to throw them. In particular, it is not advised to throw a continuation that was captured during a use: after throwing it, the system would find itself in a situation where it believes it is loading a file, but where no such file is open. A core dump is almost certain to ensue. I don’t plan to fix this soon.

- Scale syntax is kludgy, but I don’t see any way of fixing it nicely.

- Toplevel should provide a secondary prompt on incomplete input. Currently, it does not show any, which can be confusing. Also, the toplevel parser shows two prompts after successfully using a file.

- inoutprocess exhibits quirky behaviour. This seems to be due to some cruftiness inside Unix, where opening a bidirectional channel with a child process by using two pipes has strange consequences. In particular, try inoutprocess on the Unix command cat. You would think that sending the child cat process a newline-terminated line, and then reading the output from cat would give you back your message, but it won’t on most Unix machines. This is not related to flushing buffers, either in HimML or in the child process. This is unfortunate, since HimML will block on reading, deadlocking both processes. To avoid this, you should first test your own communication protocols by hand on small examples using inoutprocess.

5.8 Common Problems

5.8.1 Problems When Installing HimML

P: When I type make, nothing happens except that I get a message telling me to type a sequence of commands.

This is normal. The installation procedure needs to make configuration files, for interpreting your favorite options (in file OPTIONS) or for determining system or compiler behaviours. So, just do as indicated.

P: I don’t understand the meaning of an option in file OPTIONS.

Then leave it alone. Most options have reasonable default values.

P: When I run HimML, it just stops on abort: attempt to longjmp() to lower stack or a similar message.

See next question.

P: After typing make, I get messages such as:

mksyscc: 20847 Abort - core dumped
longjmp() is brain-damaged (won’t allow you to jump to a lower stack)
trying to find a standard patch...

Some operating systems (mostly BSD systems, although the only example I know is AIX) implement a “smart” longjmp() routine that first checks whether the current stack pointer is lower than the one it is trying to restore, and aborts if this is not the case. HimML needs to be able to do just that, in order to implement continuations (and continuations are heavily used internally, even if you don’t plan to use them). The best solution I’ve come up with on AIX is to write a small patching utility (dpxljbhak) that hunts for a specific piece of code in the prologue of the longjmp() function and puts no-ops instead. A better solution would be to rewrite the function in assembler, but I’ve been unable to do this.

If this happens to you, try to rewrite longjmp() so that it does not check for stack levels and link your new definition. Or write a patch, just like me; you’ll need to experiment a bit.

81
Please also contribute your modification so that I can include it in the next HimML release. (See MAINTENANCE at the end of the OPTIONS file to know whom to write to.)

P: My machine is a Cray/VMS machine/PC-Dos machine, and I cannot manage to make the darn thing compile or execute.

Cray machines have a weird stack format, and my scheme for capturing continuations has no hope of working on these machines. If it’s absolutely necessary for you, I’ll see what I can do, provided you promise to tell me whether it works or not. (See MAINTENANCE at the end of the OPTIONS file to know my address.)

I don’t have any VMS machine handy, so I cannot test HimML on it. The HimML implementation is pretty much centered around Unix, so I would be surprised if it worked without changes. Please tell me what you have been forced to do to make it work.

PC-Dos machines won’t do. 640K is not enough for HimML, and HimML has no knowledge of extended or expanded memory. HimML must run in one segment only, lest its sharing mechanism be defeated by one physical address having two distinct representations (from two different segments). This may work on 486’s or higher, which can use large segments, but the operating system (Dos or Windows, any version until now) is the stumbling block. Your best bet is to change for Linux or any other Unix for PCs. Windows/NT or OS/2 is expected not to pose any problem.

P: When I run HimML, it just core dumps.

Check the OPTIONS file: there is no safeguard against illegal values there (in particular stack values). Put back the default values; if this does not work, try to increase the stack parameters (notably SAFETY_SIZE and SECURITY). See also previous questions; it is quite likely that this is due to stack problems. If nothing works, mail me (goubault@lsv.ens-cachan.fr, see MAINTENANCE at the end of the OPTIONS file).

5.8.2 Problems When Running HimML

P: I have typed a command line at the toplevel prompt, then typed return, but nothing happens.

Most probably, you have not terminated your command line with a semicolon (;). Although the syntax of Standard ML makes semicolons optional between declarations, the toplevel parser has no way of knowing that input is complete unless it finds a terminating semicolon (or an end of file). Consider also all the ways to complete input such as, say, 1; if you write a semicolon afterwards, then this is an abbreviation of val it=1;, but if you write +2; even on the following line, then you really meant val it=1+2;, and if you type return just after 1, the parser has no way to know which possibility you intended.

It may happen that typing a semicolon does not cure the problem. This may happen is you have not closed all parentheses and brackets. Consider {frozzle ()}; if you type a semicolon afterwards, then your input is still incomplete, as you may want to write, say, {frozzle (); foo}. The semicolon is not only a declaration separator, but also the sequence instruction.

P: When opening modules that open header modules, I keep getting type errors, and the explanation is that some datatypes are not the same in each type?

First, check that you are not defining or declaring datatypes (or dimensions) in header files that you use instead of opening. Each time you use a given file, it creates new versions of the datatypes or dimensions inside it. To avoid it, open the file instead; this creates unique stamps for the datatype (or dimension), which it records in a file of the same name, with .mlx at the end. This will work only if your header file can be compiled separately, so be prepared to modularize your code.

If the above does not apply, it may happen that your .ml files have inconsistent modification dates. The module system always tries to recompile a .ml file when the .ml file appears to be newer than the corresponding .mlx file. Therefore, if the last modification date of the .ml file is some future date, it will always recompile it, as many times as it is opened; and this leads to the same problem as above. A quick fix is to set the modification date manually (with touch on Unix, or setdate on Amigas; there’s probably a public-domain utility to fix
this on Macintoshes, but I don’t know). In any case, there’s probably something wrong with the way the date is set up on your system, and it’s worth having a look at it.

5.9 Reporting Bugs, Making Suggestions

This is an alpha revision of HimML. This means that I do not consider it as a distributable version. This means that I deem the product robust enough to be given only to my friends, counting on their comprehensive support, mostly as far as bugs are concerned. This also means that I want some feedback on the usability of the language, and on reasonable ways to improve the implementation.

To help me improve the implementation (and possibly the language, though I am not eager to), you can submit a note to the person in charge of maintaining the system (type `#maintenance features` at the toplevel to know who, where and when). The preferred communication means is electronic mail, but others (snail-mail notably) are welcome. If you think you have found a bug in HimML, or if you want something changed in HimML, you should send the person in charge a message that should contain:

- whether it is a bug or a suggestion of improvement;
- what the problem or suggestion is. You should give it a meaningful title, and a precise description.

In case of a bug, the preferred description is in form of a short piece of code, together with the symptoms, and the kind of machine and operating system you are working on. It should be possible for somebody else than you to replay the bug. If you don’t find any small code that would exhibit the same buggy behaviour as the one you’ve just experienced, send the contents of the HimML.trace file: every time you use HimML, it logs every single toplevel or file input in this file, so as to ease replaying your actions. This may not always work, but it can help. (This file may have another name, if you have chosen to use the `-replay-file` command-line option.)

In case of a suggestion, please refrain from submitting your idea of what would be a cute extension of the language. Suggestions should improve the level of comfort you can have from using the implementation, and should be implementable without destroying the spirit of HimML. If you want to propose a suggestion, definitely argue that it will be needed, and the maintainer will try and see if it is doable.
Appendix A

Precedence Table

This is the default set of precedences when you launch HimML:

```
infix 0 before
infix 3 o :=
infix 4 = <> < > <= >= #< #> #<= #>= inset inmap submap subset strless
infixr 5 @ 0
infixr 5 ::
infix 6 + - ^ #+ #– ++ U <| <-| |> |->
infix 7 * div mod divmod #* #/ fdiv fmod fdivmod & \ delta intersects
infixr 8 #`
infix 9 nth to sub
```

Note that @ is declared infixr, that is, right-associative, although the Definition of Standard ML dictates that it is infix, i.e. left-associative. This should not make much differences.
Bibliography


Keyword index

! ......................................................... 34
#*+ ...................................................... 42
#+ ......................................................... 42
#- ......................................................... 42
#/ ........................................................ 42
#< ......................................................... 43
#<= ......................................................... 43
#> ........................................................ 43
#>= ......................................................... 43
#compile .............................................. 69,79
#* ......................................................... 43
#* ......................................................... 43
(/) .......................................................... 14
(*$D++) .................................................. 73
(*$D--*) ................................................ 73
(*$P++) .................................................. 73
(*$P--) .................................................. 73
* .......................................................... 19,39
++| ........................................................ 15
- .......................................................... 39
-- ......................................................... 69
-> .......................................................... 21
-array-hash-size ..................................... 67
-c .......................................................... 69
-col ....................................................... 68
-gctrace ............................................... 67
-grow ..................................................... 68
-init ...................................................... 67
-inline-limit ......................................... 69
-int-hash-limit ....................................... 66
-m> ........................................................ 21
-maxcells ............................................... 66
-maxgrow ............................................... 69
-mem ...................................................... 66
-nodebug ............................................... 69
-nthreads ............................................... 66
-pair-hash-size ....................................... 66
-path .................................................... 65,67
-pwd-prompt .......................................... 69
-real-hash-size ....................................... 66
-replay .................................................. 65
-string-hash-size ..................................... 67
-threadsize .......................................... 66
. .......................................................... 34
:= .......................................................... 34
:= .......................................................... 18
:= .......................................................... 34
; .......................................................... 7
< .......................................................... 40
<-| ........................................................ 28
<= .......................................................... 40
=> .......................................................... 24
<| ........................................................ 28
= .......................................................... 24
> .......................................................... 40
>= .......................................................... 40
@ .......................................................... 25
abort ...................................................... 64
abs ........................................................ 40
ach ........................................................ 44
acos ....................................................... 44
advance .................................................. 56
all ......................................................... 6
andalso .................................................... 7
app ......................................................... 26,7
append ................................................... 25
appendfile ............................................. 57
arg ........................................................ 44
args ....................................................... 60,69
Arith ...................................................... 39,42
array ...................................................... 19,35
arrayoflist ............................................ 35
Asci ....................................................... 35
ash ......................................................... 44
asin ....................................................... 44
atan ....................................................... 44
ath ......................................................... 44
b .......................................................... 72
band ....................................................... 40
before .................................................... 24
Bind ...................................................... 23
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>17,19,45</td>
</tr>
<tr>
<td>intarray</td>
<td>19</td>
</tr>
<tr>
<td>integer</td>
<td>45</td>
</tr>
<tr>
<td>inter</td>
<td>30</td>
</tr>
<tr>
<td>intersects</td>
<td>30</td>
</tr>
<tr>
<td>Intofset</td>
<td>50</td>
</tr>
<tr>
<td>intofstring</td>
<td>39</td>
</tr>
<tr>
<td>inv</td>
<td>31</td>
</tr>
<tr>
<td>IO</td>
<td>56</td>
</tr>
<tr>
<td>iomsg</td>
<td>60</td>
</tr>
<tr>
<td>irand</td>
<td>41</td>
</tr>
<tr>
<td>isub</td>
<td>35</td>
</tr>
<tr>
<td>it</td>
<td>61</td>
</tr>
<tr>
<td>iterate</td>
<td>7</td>
</tr>
<tr>
<td>iupdate</td>
<td>35</td>
</tr>
<tr>
<td>l</td>
<td>72</td>
</tr>
<tr>
<td>ldexp</td>
<td>45</td>
</tr>
<tr>
<td>leftmargin</td>
<td>61</td>
</tr>
<tr>
<td>len</td>
<td>25</td>
</tr>
<tr>
<td>length</td>
<td>35</td>
</tr>
<tr>
<td>Lip</td>
<td>45</td>
</tr>
<tr>
<td>list</td>
<td>18,72</td>
</tr>
<tr>
<td>log</td>
<td>44</td>
</tr>
<tr>
<td>logl</td>
<td>44</td>
</tr>
<tr>
<td>lsl</td>
<td>40</td>
</tr>
<tr>
<td>maintenance</td>
<td>63</td>
</tr>
<tr>
<td>major_gc</td>
<td>64</td>
</tr>
<tr>
<td>map</td>
<td>26</td>
</tr>
<tr>
<td>mapadd</td>
<td>31</td>
</tr>
<tr>
<td>mapadddunder</td>
<td>31</td>
</tr>
<tr>
<td>MapGet</td>
<td>26</td>
</tr>
<tr>
<td>mapoflist</td>
<td>32</td>
</tr>
<tr>
<td>mapremove</td>
<td>31</td>
</tr>
<tr>
<td>Match</td>
<td>23</td>
</tr>
<tr>
<td>match</td>
<td>37</td>
</tr>
<tr>
<td>matches</td>
<td>37</td>
</tr>
<tr>
<td>matchsub</td>
<td>37</td>
</tr>
<tr>
<td>max</td>
<td>40</td>
</tr>
<tr>
<td>maxprintlines</td>
<td>61</td>
</tr>
<tr>
<td>max_int</td>
<td>17,39</td>
</tr>
<tr>
<td>maybe</td>
<td>45</td>
</tr>
<tr>
<td>MDenormal</td>
<td>41</td>
</tr>
<tr>
<td>memofn</td>
<td>15</td>
</tr>
<tr>
<td>memofun</td>
<td>15</td>
</tr>
<tr>
<td>min</td>
<td>40</td>
</tr>
<tr>
<td>MInf</td>
<td>41</td>
</tr>
<tr>
<td>min_int</td>
<td>17,39</td>
</tr>
<tr>
<td>MNaN</td>
<td>41</td>
</tr>
<tr>
<td>MNormal</td>
<td>41</td>
</tr>
<tr>
<td>mod</td>
<td>40</td>
</tr>
<tr>
<td>modf</td>
<td>45</td>
</tr>
</tbody>
</table>
## Notion index

<table>
<thead>
<tr>
<th>Abstract data types</th>
<th>Accessing:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a list</td>
</tr>
<tr>
<td></td>
<td>23</td>
</tr>
<tr>
<td>Adding one maplet to a map</td>
<td>28, 29</td>
</tr>
<tr>
<td>Addition</td>
<td>12, 35, 37</td>
</tr>
<tr>
<td>Amiga</td>
<td>59</td>
</tr>
<tr>
<td>Applying a map</td>
<td>25</td>
</tr>
<tr>
<td>Arbitrary precision</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arc cosine</td>
</tr>
<tr>
<td></td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>argument</td>
</tr>
<tr>
<td></td>
<td>39</td>
</tr>
<tr>
<td>ASCII</td>
<td>31, 76</td>
</tr>
<tr>
<td>ASCII code</td>
<td>32</td>
</tr>
<tr>
<td>Assignment</td>
<td>30</td>
</tr>
<tr>
<td>Backtracking</td>
<td>9, 10</td>
</tr>
<tr>
<td>Batch session</td>
<td>61</td>
</tr>
<tr>
<td>Belongs to</td>
<td>25</td>
</tr>
<tr>
<td>Bezout’s Theorem</td>
<td>45</td>
</tr>
<tr>
<td>Bit reversal</td>
<td>44</td>
</tr>
<tr>
<td>Bit vectors</td>
<td>44</td>
</tr>
<tr>
<td>Booting</td>
<td>59</td>
</tr>
<tr>
<td>Breakpoints</td>
<td>56</td>
</tr>
<tr>
<td>Bugs</td>
<td>79</td>
</tr>
<tr>
<td>C</td>
<td>68</td>
</tr>
<tr>
<td>Cache</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in map operations</td>
</tr>
<tr>
<td></td>
<td>25</td>
</tr>
<tr>
<td>Call stack</td>
<td>56</td>
</tr>
<tr>
<td>CAML</td>
<td>18</td>
</tr>
<tr>
<td>Cardinal</td>
<td>9, 25</td>
</tr>
<tr>
<td>Ceiling</td>
<td>39</td>
</tr>
<tr>
<td>Characters: see strings</td>
<td></td>
</tr>
<tr>
<td>Checkpoints</td>
<td>18</td>
</tr>
<tr>
<td>Chinese remainder</td>
<td>45</td>
</tr>
<tr>
<td>Choice: see quantifications (choice)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24, 27</td>
</tr>
<tr>
<td>Classes</td>
<td>50</td>
</tr>
<tr>
<td>Classification of numbers</td>
<td>50, 51</td>
</tr>
<tr>
<td>Code of a character</td>
<td>36, 37</td>
</tr>
<tr>
<td>Code points</td>
<td>66</td>
</tr>
<tr>
<td>Communication</td>
<td>18</td>
</tr>
<tr>
<td>Comparison</td>
<td></td>
</tr>
<tr>
<td></td>
<td>of integers</td>
</tr>
<tr>
<td></td>
<td>35</td>
</tr>
<tr>
<td>Compilation</td>
<td>63</td>
</tr>
<tr>
<td>Complex numbers</td>
<td>11, 16, 56</td>
</tr>
<tr>
<td>Composition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>of functions</td>
</tr>
<tr>
<td></td>
<td>22</td>
</tr>
<tr>
<td>Conditional compilation</td>
<td>68</td>
</tr>
<tr>
<td>Conjugate</td>
<td>38</td>
</tr>
<tr>
<td>Conjunction</td>
<td>35, 45</td>
</tr>
<tr>
<td>Continuations</td>
<td>18, 56</td>
</tr>
<tr>
<td>Applying</td>
<td>18, 58</td>
</tr>
<tr>
<td>Reifying</td>
<td>18, 57</td>
</tr>
<tr>
<td>Converting</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a character into a number</td>
</tr>
<tr>
<td></td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
</tr>
<tr>
<td>Coroutine</td>
<td>57</td>
</tr>
<tr>
<td>Cosine</td>
<td>39</td>
</tr>
<tr>
<td>Creating</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a new array</td>
</tr>
<tr>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Creator</td>
<td>57</td>
</tr>
<tr>
<td>Cross-product</td>
<td>3</td>
</tr>
<tr>
<td>Cryptography</td>
<td>48, 49</td>
</tr>
<tr>
<td>Current file position</td>
<td>51</td>
</tr>
<tr>
<td>Datatypes: see types (datatypes)</td>
<td></td>
</tr>
<tr>
<td>Debugging</td>
<td>56, 64</td>
</tr>
</tbody>
</table>
decrementing ........................................ 36
delaying ............................................. 18, 57
denormalized numbers ............................ 36, 37
dependency ........................................... 63
dereferencing ....................................... 30
   a reference ....................................... 30
   an element of an array .......................... 31
difference of sets ................................... 28
dimensions .......................................... 11, 15
disjunction .......................................... 35, 45
division .............................................. 12, 35, 37, 39, 40
domain description ................................. 3
   ranging over elements of a list ............... 3
   ranging over elements of a set ............... 3
   ranging over maplets in a map ............... 3
   ranging over submaps of a map ............... 3
   traversal order .................................. 3, 4, 5
domain of a map .................................... 24, 25, 28
domain restriction ................................. 25, 26, 28

dynamics ........................................... 18, 51
editor ................................................. 76
elaboration : see typing elements
   of a list .......................................... 30
   of a map .......................................... 25
   of a set .......................................... 25
ellipsis ............................................. 3
   in record patterns ............................... 19
   in record types .................................. 19
   in set patterns ................................... 9
   in tuple patterns ................................ 18
   in tuple types .................................. 18, 19
elliptic curve method ............................. 50
Emacs mode ......................................... 76
empty map .......................................... 10, 24
empty record ....................................... 13
empty set : see empty record
empty tuple ......................................... 13
equality ............................................ 22
   admitting equality .............................. 19
   admitting equality : 3, 10, 11, 15, 18, 19, 20, 22, 30
   hidden uses ...................................... 37
   of numbers ....................................... 37
equality declaration .............................. 20
equality types : see equality (admitting equality)
Euclidean algorithms ............................ 45
exception handlers ............................... 57
exceptions .......................................... 18, 21, 64
exclusive or ........................................ 35, 45
executable specification : see specification
exhaustive patterns .............................. 21
existential quantification : see quantifications (existential)
exlosion of a string ................................ 31
exponential ......................................... 38
exponentiation : see power operation
extended gcd ....................................... 45
extendible types : see types (tuple and record)
extension of types .................................. 13, 19
factoring
   by Pollard’s ρ algorithm ......................... 48
   factoring integers ................................ 50
   feature record .................................... 56
   field selector : see selector
   fields ............................................... 19
   numerical .......................................... 13, 19
files .................................................. 13, 51, 52
filter ................................................... 3, 4
floating-point numbers .......................... 11, 16, 36, 56
floor ................................................... 39
flushing a stream .................................. 50
forcing a promise ................................. 18, 57
functions ............................................ 20
garbage collection .................................. 21, 58, 59, 62
Gcd .................................................... 45
gcd ..................................................... 45
getting an element of a list ..................... 23
GNU Emacs ........................................... 76
Greatest common divisor ......................... 45
greatest common divisor ......................... 45
grep ................................................... 33
header file .......................................... 73
himml-mode.el ....................................... 76
HimMLtrace .......................................... 59
hyperbolic functions ............................. 39
IEEE754 .............................................. 36, 56
image of an element by a map ................... 25
imaginary part ...................................... 38
imperative comprehension ........................ 6
imperative types .................................... 11, 18, 56
implosion of a string ............................. 32
importing ............................................ 71
improving the implementation .................... 79
inclusion
   of maps .......................................... 25
   of sets ............................................ 25
incremental functions ............................ 27
incrementing ....................................... 35
indices of a list .................................... 29
infinity .............................................. 36
inheritance
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiple inheritance</td>
<td>13, 19, 20</td>
</tr>
<tr>
<td>single inheritance</td>
<td>13, 19</td>
</tr>
<tr>
<td>initialization</td>
<td>61</td>
</tr>
<tr>
<td>input : see streams</td>
<td></td>
</tr>
<tr>
<td>input/output</td>
<td>13, 50, 61</td>
</tr>
<tr>
<td>integer part</td>
<td></td>
</tr>
<tr>
<td>lower</td>
<td>39</td>
</tr>
<tr>
<td>upper</td>
<td>39</td>
</tr>
<tr>
<td>integers</td>
<td>40</td>
</tr>
<tr>
<td>interleaving : see domain description (traversal order)</td>
<td></td>
</tr>
<tr>
<td>intersection</td>
<td>24, 28, 45</td>
</tr>
<tr>
<td>interval</td>
<td>see range of numbers</td>
</tr>
<tr>
<td>inverse of a map</td>
<td>29</td>
</tr>
<tr>
<td>iteration : see quantifications (iteration)</td>
<td></td>
</tr>
<tr>
<td>Jacobi symbol</td>
<td>43</td>
</tr>
<tr>
<td>Jean Goubault</td>
<td>57</td>
</tr>
<tr>
<td>labels</td>
<td>13, 19</td>
</tr>
<tr>
<td>large integer</td>
<td>40</td>
</tr>
<tr>
<td>last toplevel expression</td>
<td>55</td>
</tr>
<tr>
<td>lazy data structures</td>
<td>18</td>
</tr>
<tr>
<td>leak (space)</td>
<td>58</td>
</tr>
<tr>
<td>leaving a session</td>
<td>58</td>
</tr>
<tr>
<td>Legendre symbol</td>
<td>43</td>
</tr>
<tr>
<td>length</td>
<td></td>
</tr>
<tr>
<td>of a list</td>
<td>23</td>
</tr>
<tr>
<td>of a string</td>
<td>31, 32</td>
</tr>
<tr>
<td>of an array</td>
<td>31</td>
</tr>
<tr>
<td>Lenstra, Arjen</td>
<td>40, 50</td>
</tr>
<tr>
<td>linking</td>
<td>63</td>
</tr>
<tr>
<td>linking a module</td>
<td>72</td>
</tr>
<tr>
<td>lists</td>
<td>20, 22</td>
</tr>
<tr>
<td>defined by comprehension</td>
<td>5, 21, 23, 26</td>
</tr>
<tr>
<td>defined by imperative comprehension</td>
<td>7</td>
</tr>
<tr>
<td>load path</td>
<td>61</td>
</tr>
<tr>
<td>loading a file</td>
<td>59</td>
</tr>
<tr>
<td>logarithm</td>
<td>39, 43</td>
</tr>
<tr>
<td>base 2</td>
<td>44</td>
</tr>
<tr>
<td>Macintosh</td>
<td>59</td>
</tr>
<tr>
<td>maintenance</td>
<td>57, 79</td>
</tr>
<tr>
<td>make</td>
<td>63, 75</td>
</tr>
<tr>
<td>makefile</td>
<td>63</td>
</tr>
<tr>
<td>maplets</td>
<td>2</td>
</tr>
<tr>
<td>maps : see sets</td>
<td></td>
</tr>
<tr>
<td>maximum</td>
<td>35, 38</td>
</tr>
<tr>
<td>measure units : see units</td>
<td></td>
</tr>
<tr>
<td>memoization</td>
<td>2, 14, 27, 57</td>
</tr>
<tr>
<td>as an alternative to promises</td>
<td>18</td>
</tr>
<tr>
<td>memory usage policy</td>
<td>60, 62</td>
</tr>
<tr>
<td>methods</td>
<td>50</td>
</tr>
<tr>
<td>Miller-Rabin test</td>
<td>48</td>
</tr>
<tr>
<td>minimum</td>
<td>35, 38</td>
</tr>
<tr>
<td>ml-mode.el</td>
<td>76</td>
</tr>
<tr>
<td>modifying a reference</td>
<td>30</td>
</tr>
<tr>
<td>modifying an array</td>
<td>31</td>
</tr>
<tr>
<td>modular arithmetic</td>
<td></td>
</tr>
<tr>
<td>Montgomery</td>
<td>40, 46</td>
</tr>
<tr>
<td>standard</td>
<td>45</td>
</tr>
<tr>
<td>modules</td>
<td>2, 56</td>
</tr>
<tr>
<td>modulo</td>
<td>35, 39, 40</td>
</tr>
<tr>
<td>Montgomery</td>
<td>40, 46</td>
</tr>
<tr>
<td>multiplication</td>
<td>35, 37</td>
</tr>
<tr>
<td>of numerical types</td>
<td>11, 15, 20</td>
</tr>
<tr>
<td>of scales (units)</td>
<td>12, 20</td>
</tr>
<tr>
<td>mutable : see references, arrays</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>36, 38</td>
</tr>
<tr>
<td>natural numbers</td>
<td>40</td>
</tr>
<tr>
<td>negation</td>
<td>35, 45</td>
</tr>
<tr>
<td>integer</td>
<td>35</td>
</tr>
<tr>
<td>logical</td>
<td>22</td>
</tr>
<tr>
<td>numerical</td>
<td>38</td>
</tr>
<tr>
<td>nondeterminism</td>
<td>3, 4, 5</td>
</tr>
<tr>
<td>Not a Number : see NaN</td>
<td></td>
</tr>
<tr>
<td>numerical labels : see fields (numerical)</td>
<td></td>
</tr>
<tr>
<td>numerical types : see types (numerical)</td>
<td></td>
</tr>
<tr>
<td>numerical values</td>
<td>2, 11, 12</td>
</tr>
<tr>
<td>object module</td>
<td>72</td>
</tr>
<tr>
<td>opening a file</td>
<td>51, 52</td>
</tr>
<tr>
<td>opening a module</td>
<td>72</td>
</tr>
<tr>
<td>operating system</td>
<td>56, 58</td>
</tr>
<tr>
<td>order of traversal : see domain description</td>
<td></td>
</tr>
<tr>
<td>output : see streams</td>
<td></td>
</tr>
<tr>
<td>overwriting</td>
<td>4, 6, 26, 27, 28</td>
</tr>
<tr>
<td>patterns</td>
<td></td>
</tr>
<tr>
<td>records</td>
<td>19</td>
</tr>
<tr>
<td>sets and maps</td>
<td>4, 9</td>
</tr>
<tr>
<td>tuples</td>
<td>18</td>
</tr>
<tr>
<td>physical units : see units</td>
<td></td>
</tr>
<tr>
<td>π</td>
<td>38</td>
</tr>
<tr>
<td>Pollard</td>
<td></td>
</tr>
<tr>
<td>ρ algorithm</td>
<td>48</td>
</tr>
<tr>
<td>polymorphism</td>
<td>12, 15, 18</td>
</tr>
<tr>
<td>in printing</td>
<td>18</td>
</tr>
<tr>
<td>portability</td>
<td>68</td>
</tr>
<tr>
<td>positioning a file</td>
<td>50, 51</td>
</tr>
<tr>
<td>power</td>
<td></td>
</tr>
<tr>
<td>of numerical types</td>
<td>11, 15, 20</td>
</tr>
<tr>
<td>of scales (units)</td>
<td>12, 20</td>
</tr>
<tr>
<td>power operation</td>
<td>12, 38</td>
</tr>
<tr>
<td>precompiling modules</td>
<td>72</td>
</tr>
<tr>
<td>preprocessor</td>
<td>68</td>
</tr>
<tr>
<td>pretty-printing : see printing</td>
<td></td>
</tr>
<tr>
<td>prime</td>
<td></td>
</tr>
</tbody>
</table>
number ........................................ 48
random ........................................ 48
prime number ................................. 48
generation .................................... 48
printing ........................................ 51
to a string .................................... 50, 53
product of types : see types (product)
profiling ....................................... 56, 67
promises ....................................... 18, 57
prompt .......................................... 64
pseudo-random booleans ................... 40
pseudo-random numbers ................. 36, 40, 48
quantifications .............................. 5, 21
choice .......................................... 6
existential .................................... 5, 6
existential imperative ..................... 8
imperative choice ............................ 8
imperative iteration ....................... 8
iteration ....................................... 6
universal ..................................... 5
universal imperative ....................... 7
quitting ....................................... 58
quotient ....................................... 35, 39, 40
random booleans ............................ 40
random numbers ......................... 36, 40, 48
random primes ............................... 48
range of a map ............................. 25
range of numbers ........................... 24, 29
range restriction ......................... 26
reading ................................. 50, 53
from a string .................................. 50
reading a line .................................. 51
real numbers .................................. 37, 40
real part ....................................... 38
recompilation (last) ....................... 57
record .......................................... 20
records ....................................... 19, 50
delimiters ..................................... 12, 19
patterns ....................................... 19
redundant patterns ....................... 21
references ..................................... 17, 30
regular expression ....................... 33
remainder .................................... 35, 39, 40
removing an element from a map ...... 29
replay debugger ............................. 56
replay file .................................... 59
replay-file ................................... 79
reversal ....................................... 44
of bits ........................................ 44
reverse execution ........................... 56
root ........................................... 43
square ........................................ 38, 43
rounding ..................................... 36, 38
run-time architecture ..................... 2
maximal sharing ............................ 2
scales : see units ............................ 18
Scheme ......................................... 50
seeking ........................................ 50, 51
selector ....................................... 18, 19, 20
separate compilation ..................... 71
sequencing ................................. 6, 22
sets ........................................... 2, 3, 14, 19, 20, 24
cross-product traversal .................. 4
defined by comprehension ............ 3, 4, 21, 28
defined by enumeration .................. 3
defined by imperative comprehension .. 6
map type constructor ..................... 3
need for ...................................... 2
parallel traversal ......................... 4
patterns ....................................... 4, 9
set type constructor ..................... 3
typing ......................................... 10
side-effects ................................. 6
in imperative comprehensions ........ 6
in iterations ................................. 6
sign of a number ............................ 36
sine ........................................... 39
size : see length ............................ 10
SML–NJ : see Standard ML of New Jersey
space leak ..................................... 58
specification .................................... 2
splitting ....................................... 9, 27
square root .................................... 38, 43
squaring ....................................... 35, 38
stack ........................................... 21, 56
standard error .................................. 51
standard input .................................. 52, 61
Standard ML 2, 3, 4, 5, 6, 7, 9, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, 23, 30, 56, 58
Standard ML of New Jersey 2, 13, 15, 17, 23, 30, 56, 76
standard output ............................. 51
starting a session ........................... 59
starting time .................................... 57
stepping ....................................... 65
streams ................................. 50, 50
input ........................................... 50
output ......................................... 50
strength ....................................... 15, 16, 18, 19
do a type expression .................... 15, 16, 18, 19
of a type variable ......................... 11, 15
strings ......................................... 16, 31