A Hope Interpreter — Reference

Ross Paterson

April 18, 2000

This manual is not a tutorial on functional programming, or on the language HOPE. If you don't know about both, you might start with something like Roger Bailey's tutorial [1].

1 Lexical structure

The input is divided into a sequence of symbols of the following kinds:

- a punctuation character: one of "`( ) , ; [ ]`",
- an identifier: either
  1. a letter or underscore followed by zero or more letters, underscores or digits (followed by zero or more single quotes), or
  2. a sequence of graphic characters that are neither white-space, letters, underscores, digits, punctuation nor one of "`! " '`",
- a numeric literal: a sequence of digits (on some systems, optionally with an embedded decimal point and an optionally signed exponent),
- a character literal: either a single character (but not a newline), or a character escape sequence (as in the C language), enclosed in single quotes, or
- a string literal: a sequence of zero or more characters (but not newlines of double quotes) or character escape sequences, enclosed in double quotes.

Symbols may be separated by white-space and comments. A comment is introduced by '`!`' and continues to the end of the line.

The following identifiers are reserved by the language:

```
++ --- : <= == => |
abstype data dec display else edit exit if in
infix infixr lambda let letrec private save then
type typevar uses where whererec write
```

The following identifers are also reserved, for compatability with other implementations:

```
end module nonop pubconst pubfun pubtype
```

Also, the following synonyms are available: `\` for `lambda`, `use` for `uses` and `infixrl` for `infixr`.

2 Identifiers

Identifiers may refer to

- modules,
- types,
- type constructors,
- data constants,
- data constructors, or
- values.

The same identifier may refer to more than one of these kinds, but in any scope it cannot refer to more than one of the last three classes.

Identifiers may be defined as *infix operators*, by `infix` and `infixr` definitions (see section 3.2). This affects only the way in which they are parsed and printed: if an identifier $\oplus$ is declared as an infix operator, constructs of the form $\oplus$`(`*a*`, `*b*`)` are written *a* $\oplus$ *b*. To refer to $\oplus$ on its own, use `(`$\oplus$`)`. A special case: in any subsequent declaration of $\oplus$ as a data constructor, $\oplus$`(`*type*$_1$ `#` *type*$_2$`)` is written *type*$_1$ $\oplus$ *type*$_2$.

A number of identifiers are predefined in the module `Standard` (see appendix A).

3 Composite structures

In the following description, constant width text (e.g. `dec` or `++`) denotes literal program text, while italic text (e.g. *type-identifier* or *pattern*) denotes syntactic classes. Non-terminals are defined in sections of the same name. The brackets [and] are used around optional text. Alternative forms are used if identifiers are declared as operators (see section 2).

3.1 Modules

A Hope module *name* is a text file *name*`.hop` consisting of definitions. The definitions may occur in any order, but identifiers must be appropriately declared before use.

There is a special Hope module called `Standard` (see appendix A), which is implicitly imported by every other Hope module and session.

An interactive session consists of definitions (see section 3.2) together with other commands (see section 3.6).

3.2 Definitions

`uses` *module-identifier*`, ..., `*module-identifier*`;`
: all the definitions visible in the named modules are to be visible in the current module or session. The modules are searched for first in the current directory, and then in a library directory. Used modules may use other modules, and so on, provided there are no cycles. Definitions visible in the using module (or session) will not be visible in a used module, unless the definitions reside in a third module used by both.

`private;`

all subsequent definitions in this module will be hidden from other modules using it. In particular, the visible definitions in a module used after this directive will not be visible to any module using this one, whereas they would be if the module was used before the directive.

`infix` *identifier*`, ...,` *identifier* `:` *numeric-literal*`;`

declare left associative infix operators with the stated precedence, from `1` (weakest) to `9` (strongest).

`infixr` *identifier*`, ...,` *identifier* `:` *numeric-literal*`;`

like `infix`, except that the operators are right associative.

`abstype` *identifier* `[(`$identifier_1$`, ...,` $identifier_n$`)];`

an 'abstract' type definition, declaring the *identifier* as a type or type constructor identifier, which may be used in subsequent definitions. The *identifier* may be defined later by a `data` or `type` definition. If there is only one argument, the parentheses may be omitted.

`data` *identifier* `[(`$identifier_1$`, ...,` $identifier_n$`)] ==`
$identifier'_1$ `[`$type_1$`]` `++ ... ++` $identifier'_k$ `[`$type_k$`];`

define the *identifier* as a type or type constructor, with the $identifier'_k$ as its data constants and constructors. If there is only one simple argument, the parentheses may be omitted.

The definition may be recursive. Any uses of *identifier* in $type_i$ should have the same parameters as the left-hand side.

Any constructor identifier may have been previously declared in a `dec` declaration, in which case the new type of the identifier must be an instance of that given in the prior declaration.

`type` *identifier* `[(`$identifier_1$`, ...,` $identifier_n$`)] ==` *type*`;`

define the *identifier* as an abbreviation for the *type*, which may refer to the argument type identifiers. If there is only one simple argument, the parentheses may be omitted.

The definition may be recursive. Any uses of *identifier* in *type* should have the same parameters as the left-hand side. See section B.1 for more details.

`typevar` *identifier*`, ...,` *identifier*`;`

declare new type variables, for use in `dec` declarations.

If *identifier* is declared, then *identifier*`'`, *identifier*`''` and so on may also be used as type variables (single quotes in *identifier* itself are ignored).

`dec` *identifier*`, ...,` *identifier* `:` *type*`;`

declare the identifiers as value identifiers of the given type, in which type variables declared by `typevar` definitions stand for any type.

If only one identifier is being declared, the keyword `dec` is optional.

`---` *value-identifier* $pattern_1 \cdots pattern_n$ `<=` *expression*`;`

define the value of an identifier. If n is zero, only one such definition is allowed for each identifier. Otherwise, the identifier may be defined by one or more such definitions, each with the same number of arguments. See section 4 for the semantics.

The keyword `---` is optional.

3.3 Types

type-identifier
: the type referred to.

type-constructor-identifier`(`*type*`, ..., `*type*`)`
: a constructed type or type abbreviation. If there is only one type argument, the parentheses may be omitted.

`(`*type*`)`
: same as *type*.

3.4 Patterns

identifier
: (but not a *data-constant-identifier*) a variable, which matches any value. No variable may occur twice in the same pattern. Each free occurrence of the *identifier* in the expression corresponding to the pattern is bound by this pattern. That is, the identifier is a *value-identifier* in the expression, referring to the value matched.

data-constant-identifier
: matches the named data constant.

data-constructor-identifier pattern
: matches a value formed by applying the data constructor to a value matching *pattern*.

numeric-literal
: `1`, `2`, ... are abbreviations for `succ(0)`, `succ(succ(0))` etc.

pattern `+` *numeric-literal*
: The form *pattern*`+`*k* is an abbreviation for `succ` applied k times to *pattern*.

`'`*c*`'`
: matches the character constant.

`"`*string*`"`
: abbreviation for a list of characters.

`[`*pattern*$_1$`, ..., `*pattern*$_n$`]`
: equivalent to '*pattern*$_1$ `::` ... `::` *pattern*$_n$ `::` `nil`'.

`[]`
: equivalent to '`nil`'.

pattern$_1$`,` *pattern*$_2$
: matches a pair of values matching *pattern*$_1$ and *pattern*$_2$ respectively.

`(`*pattern*`)`
: same as *pattern*.

3.5 Expressions

value-identifier
: the value bound to the identifier.

data-constant-identifier
: a data constant.

data-constructor-identifier
: a data constructor.

numeric-literal
: `1`, `2`, ... are abbreviations for `succ(0)`, `succ(succ(0))` etc.

'*c*'
: character constant.

"*string*"
: abbreviation for a list of characters.

[*expression*$_1$`,` ..., *expression*$_n$]
: equivalent to '*expression*$_1$ `::` ... `::` *expression*$_n$ `::` `nil`'.

`[]`
: equivalent to '`nil`'.

expression$_1$`,` *expression*$_2$
: a pair formed from the values of *expression*$_1$ and *expression*$_2$.

`(`*expression*`)`
: same as *expression*.

expression$_1$ *expression*$_2$
: the result of applying the function or constructor value of *expression*$_1$ to the value of *expression*$_2$.

`lambda` *pattern*$_1$ `=>` *expression*$_1$ `|` $\cdots$ `|` *pattern*$_k$ `=>` *expression*$_k$
: an anonymous function. See section 4 for the semantics.

`if` *expression*$_1$ `then` *expression*$_2$ `else` *expression*$_3$
: a conditional expression, equal to *expression*$_2$ if *expression*$_1$ is `true`, or *expression*$_3$ if it is `false`.

`let` *pattern* `==` *expression*$_1$ `in` *expression*$_2$
: the same as *expression*$_2$, with the variables in *expression*$_2$ replaced by the values assigned to them in matching *pattern* to *expression*$_1$. It is equivalent to

    `(lambda` *pattern* `=>` *expression*$_2$`)` *expression*$_1$

`letrec` *pattern* `==` *expression*$_1$ `in` *expression*$_2$
: like `let`, except that *pattern* must be irrefutable, and its variables may also appear in *expression*$_1$. It is a more efficient version of

    `(lambda` *pattern* `=>` *expression*$_2$`)(fix(lambda` *pattern* `=>` *expression*$_1$`))`

  for a function `fix` defined as

```
dec fix: (alpha -> alpha) -> alpha;
--- fix f <= f(fix f);
```

expression$_1$ `where` *pattern* `==` *expression*$_2$
: same as '`let` *pattern* `==` *expression*$_2$ `in` *expression*$_1$'.

expression$_1$ `whererec` *pattern* `==` *expression*$_2$
: same as '`letrec` *pattern* `==` *expression*$_2$ `in` *expression*$_1$'.

Ambiguities in patterns and expressions are resolved by the following binding precedences, from weakest to strongest:

- comma (right associative)
- `lambda`
- `let` and `letrec`
- `where` and `whererec`
- `if`
- infix operators of precedence 1to 9
- function application (left associative)

For any infix operators $\oplus$ and expression e, the following abbreviations, called *operator sections*, are permitted:

`(`e $\oplus$`)` is short for `lambda x =>` e $\oplus$ `x`.

`(`$\oplus$ e`)` is short for `lambda x => x` $\oplus$ e.

3.6 Interactive commands

In a HOPE session, any definitions (see section 3.2) may be entered (although `private` will be ignored), as well as the following commands:

expression`;`
: display the value and most general type of *expression*. Lazy evaluation is used, but nothing is displayed until the value is fully evaluated.

  The *expression* may involve the special variable `input`, which denotes the list of characters typed at the terminal.

`write` *expression* [`to "`*file*`"`]`;`
: output the value of *expression* (which must be a list of values) more directly: if the value is a list of characters, the characters are printed; otherwise each value is printed on a line by itself. Each element of the list is printed as soon as its value is computed (by lazy evaluation). If the `to` clause is present, the output is written to the file; otherwise it is printed on the screen. It is safe to write to a file that is read by the expression—no interference will occur.

  The *expression* may involve the special variable `input`, which denotes the list of characters typed at the terminal.

`display;`
: display the definitions of the current session.

`save` *module-identifier*`;`
: save the definitions of the current session in a new module of the given name. The definitions are replaced in the current session by a reference to the new module.

`edit` [*module-identifier*]`;`
: (Unix only) invoke the default editor on the named module, if given, or otherwise on a file containing the current definitions, and then re-enter the interpreter with the revised definitions.

`exit;`
: exit from the interpreter.

4 Semantics of pattern matching

The following is a partial specification; nothing more is guaranteed. Informally, when patterns overlap, more specific patterns are preferred to more specific ones. Apart from that, the choice is unspecified. More formally:

Matching a value with a pattern involves evaluating the value sufficiently to compare its data constants and constructors with those in the pattern, in some consistent, top-to-bottom (but otherwise unspecified) order. This match may result in

success: the value can be seen to be an instance of the pattern,

failure: the value cannot be an instance of the pattern, because it has a different data constant or constructor in some position, or

non-termination: the value cannot be sufficiently evaluated to decide between the above, in the particular order of checking used.

If the match succeeds, it supplies values for the variables of the pattern.

A pattern (or sub-pattern) consisting only of variables and pairs always matches a value of the appropriate type, even if computation of the value would not terminate. Such patterns are called *irrefutable*.

Now suppose a function has been defined by a series of definitions

$$\texttt{---}\ f\ \ p_{i1}\ \cdots\ p_{in}\ \texttt{<=}\ e_i\texttt{;}$$

for $i = 1, \ldots, k$, or is the expression

$$\texttt{lambda}\ p_{11}\ \cdots\ p_{1n}\ \texttt{=>}\ e_1\ \texttt{|}\ \cdots\ \texttt{|}\ p_{k1}\ \cdots\ p_{kn}\ \texttt{=>}\ e_k$$

(At present, anonymous functions are required to be unary.) An application of this function to n argument values will have zero or more possible values, as follows:

- If some patterns p_{ij} successfully match the corresponding arguments for some i, and all more specific patterns (if any) fail, then a possible value is the value of e_i, with the variables of p_{ij} taking the values they matched.
- If the matching of some pattern does not terminate, under some order of checking, then non-termination is a possible value.

If there are no possible values, the value of the application is a run-time error. Otherwise, one of the possible values is chosen, in a deterministic (but otherwise unspecified) manner.

References

[1] Roger Bailey. A Hope tutorial. *Byte*, pages 235–258, August 1985.

[2] Roger Bailey. *Functional Programming in Hope*. Ellis Horwood, Chichester, England, 1990.

[3] R.M. Burstall, D.B MacQueen, and D.T. Sanella. Hope: An experimental applicative language. In *The 1980 LISP Conference*, pages 136–143, Stanford, August 1980. Also CSR-62-80, Dept of Computer Science, University of Edinburgh.

[4] A.J. Field and P.G. Harrison. *Functional Programming*. Addison Wesley, Wokingham, England, 1988.

A Appendix: The Standard module

The standard environment for Hope, implicitly used by every session and module.

A.1 Standard type constructors

Standard type variables.

```
typevar alpha, beta, gamma;
```

Function and product types.

```
infixr -> : 2;
abstype neg -> pos;

infixr # : 4;
abstype pos # pos;

--- (a # b) (x, y) <= (a x, b y);

infixr X : 4;
type alpha X beta == alpha # beta;
```

Normal right-to-left composition of f and g.

```
infix o : 2;
dec o : (beta -> gamma) # (alpha -> beta) -> alpha -> gamma;
--- (f o g) x <= f(g x);

--- (a -> b) f <= b o f o a;
```

The identity function.

```
dec id : alpha -> alpha;
--- id x <= x;
```

Booleans with the standard operations.

```
data bool == false ++ true;
type truval == bool;

infix or : 1;
infix and : 2;

dec not : bool -> bool;
--- not true <= false;
--- not false <= true;

dec and : bool # bool -> bool;
--- false and p <= false;
--- true and p <= p;
```

```
dec or : bool # bool -> bool;
--- true or p <= true;
--- false or p <= p;

dec if_then_else : bool -> alpha -> alpha -> alpha;
--- if true then x else y <= x;
--- if false then x else y <= y;
```

Lists.

```
infixr :: : 5;
data list alpha == nil ++ alpha :: list alpha;
```

List concatenation.

```
infixr <> : 5;
dec <> : list alpha # list alpha -> list alpha;
--- [] <> ys <= ys;
--- (x::xs) <> ys <= x::(xs <> ys);
```

The type num behaves as if it were defined by:

```
!        data num == 0 ++ succ num;
```

but is actually implemented a little more efficiently. The type also contains negative numbers (and reals in some implementations).

```
data num == succ num;
```

Similarly, the type char behaves as if it were defined as an enumeration (in the normal order) of all the ASCII character constants.

```
abstype char;

--- char x <= x;
```

A.2 Internally defined functions

Comparison functions: these evaluate their arguments only to the extent necessary to determine their order. In comparing distinct constants and constructors, the lesser is the one which came earlier in the data definition in which both were defined. Pairs compare lexicographically, while comparison of functions triggers a run-time error. On the types num, char and list(char), these rules give the normal orderings.

```
infix =, /= : 3;
infix <, =<, >, >= : 4;
dec =, /=, <, =<, >, >= : alpha # alpha -> bool;
```

Lower-level comparison functions.

```
data relation == LESS ++ EQUAL ++ GREATER;

dec compare : alpha # alpha -> relation;

--- x =  y <= (\ EQUAL => true | _ => false) (compare(x, y));
--- x /= y <= (\ EQUAL => false | _ => true) (compare(x, y));
--- x <  y <= (\ LESS => true | _ => false) (compare(x, y));
--- x >= y <= (\ LESS => false | _ => true) (compare(x, y));
--- x >  y <= (\ GREATER => true | _ => false) (compare(x, y));
--- x =< y <= (\ GREATER => false | _ => true) (compare(x, y));
```

Conversions.

```
dec ord : char -> num;
dec chr : num -> char;

dec num2str : num -> list char;
dec str2num : list char -> num;
```

The contents of a named file (created lazily).

```
dec read : list char -> list char;
```

Abort execution with an error message.

```
dec error : list char -> alpha;
```

The usual arithmetical functions.

```
infix +, - : 5;
infix *, /, div, mod : 6;
dec +, -, *, /, div, mod : num # num -> num;
```

Math library.

```
dec cos, sin, tan, acos, asin, atan : num -> num;
dec atan2, hypot : num # num -> num;
dec cosh, sinh, tanh, acosh, asinh, atanh : num -> num;
dec abs, ceil, floor : num -> num;
dec exp, log, log10, sqrt : num -> num;
dec pow : num # num -> num;
dec erf, erfc : num -> num;
```

Any extra arguments on the interpreter command line.

```
dec argv : list(list char);
```

Part of the special treatment of the succ constructor.

```
dec succ : num -> num;
--- succ n <= n+1;
```

B Deviations from other versions of the language

The following features are special to this version of HOPE: irrefutable patterns, operator sections, functions of more than one argument, recursive `type` definitions, `input`, `read`, `write`, `private`, `abstype`, `letrec` and `whererec`. Other versions may not support full lazy evaluation, and may have several features not provided here.

This version also supports curried type and data constructors.

The rest of this appendix describes some experimental features of this version of the language.

B.1 Regular types

The type system is generalized to permit *regular* types, which are possibly infinite types that are unrollings of finite trees. For example, this makes possible a definition of Curry's Y combinator:

```
dec Ycurry : (alpha -> alpha) -> alpha;
--- Ycurry f <= Z Z where Z == lambda z => f(z z);
```

This fails in the usual type system because `Z` must have a function type with argument type the same as the whole type.

Type synonyms are now allowed to be recursive, so they can be used to describe infinite types, like the type of infinite sequences:

```
type seq alpha == alpha # seq alpha;
```

Recursive uses of the type constructor being defined must have exactly the same arguments as the left-hand side. However, these arguments may be permuted, as in

```
type alternate alpha beta == alpha # alternate beta alpha;
```

The syntax of types is also extended to express regular types. For example, the above two types could be defined as

```
type seq alpha == mu s => alpha # s;
type alternate alpha beta == mu a => alpha # (beta # a);
```

This notation is also used by the system to print regular types.

B.2 Functors

Each `data` or `type` definition introduces a new 'map' function (or functor, for category theorists) with the same name and arity as the type constructor. For example, the type definition

```
data tree alpha == Empty ++ Node (tree alpha) alpha (tree alpha);
```

also defines a function

```
tree : (alpha -> beta) -> tree alpha -> tree beta
```

that maps a function over trees. This automatic definition may be explicitly overridden.

It's a little more complicated: if the type argument is used negatively, as in

```
type cond alpha == alpha -> bool;
```

the function will have a flipped type:

```
cond : (alpha -> beta) -> tree beta -> tree alpha
```

If the argument is used both positively and negatively, as in

```
type endo alpha == alpha -> alpha;
```

the function will have a type like

```
endo : (alpha -> alpha) -> tree alpha -> tree alpha
```

Similarly, an `abstype` definition declares this corresponding function. In order to determine the type, each argument is assumed to be used both positively and negatively unless the argument variable is replaced with one of the special keywords `pos`, `neg` or `none` (indicating that the argument is unused). See the previous appendix for some examples.