Andrew D. Gordon* and Kevin Hammond†

June 5, 1995

Abstract

We describe the design and use of monadic I/O in Haskell 1.3, the latest revision of the lazy functional programming language Haskell. Haskell 1.3 standardises the monadic I/O mechanisms now available in many Haskell systems. The new facilities allow more sophisticated text-based application programs to be written portably in Haskell. Apart from the use of monads, the main advances over standard Haskell 1.2 are: character I/O based on handles (analogous to ANSI C file pointers), an error handling mechanism, terminal interrupt handling and a POSIX interface. The standard also provides implementors with a flexible framework for extending Haskell to incorporate new language features. In addition to a tutorial description of the new facilities this paper includes a worked example: a monad for combinator parsing which is based on the standard I/O monad.

1 Introduction

Haskell 1.3 improves on previous versions of Haskell [11] by adopting an I/O mechanism based on *monads* [18]. This paper explains the structure of this monadic I/O mechanism, justifies some of the design decisions, and explains how to program with the new facilities. This paper provides a more in-depth treatment of I/O than is possible in the Haskell 1.3 report [8] and library documentation [9].

Previous versions of Haskell used *synchronised streams* or *dialogues* for I/O. In practice, many Haskell programmers found it cumbersome to use these constructs directly. Awkward pattern matching against the input stream was necessary, as illustrated by the program in Figure 1, which simply copies its standard input to its standard output. Instead, it was common practice to program at a higher level using libraries of derived functions. One such library (for *continuation-passing* I/O [13, 16]) used to be part of the Haskell standard prelude.

```
main ~( Str input : ~ (Success : _ )) =
  [ ReadChan stdin,
    AppendChan stdout input
  ]
```

Figure 1: Dialogue I/O in Haskell 1.2

Recently, researchers have experimented with new I/O combinators based on monads [7, 17]. These combinators are capable of capturing all the I/O operations that could be provided using the previous stream-based approach, and provide the same type security as the continuation library. The monadic approach is significantly more flexible than the other two approaches, however, in the ease with which new I/O primitives can be introduced or existing I/O primitives combined to create new combinators. Monadic I/O has proved sufficiently attractive that several Haskell systems already support at least a basic implementation, and some also support more sophisticated mechanisms such as inter-language working, concurrency, or direct state-manipulation.

One of the main purposes of Haskell 1.3 is to standardise primitives for monadic I/O. The design provides a basic (but "industrial-strength" and extensible) interface to common operating systems such as Unix, DOS, VMS, or the Macintosh. The design has been influenced by the I/O operations found in imperative languages. Experimental features with which the Haskell com-

*University of Cambridge Computer Laboratory, New Museums Site, Cambridge, CB2 3QG, UK.

†Department of Computing Science, University of Glasgow, 17 Lilybank Gdns., Glasgow, G12 8QQ, UK.

in previous versions of Haskell. Some rarely-used features, such as `Binary` files, have been removed, pending better designs.

The definition of Haskell 1.3 consists of two documents. The report proper [8] defines the Haskell language and the standard prelude. The standard libraries are defined in a separate document [9]. Sections 2, 3 and 4 of this paper describe the contents of the I/O libraries. Section 5 shows how to write combinator parsers in terms Haskell 1.3 I/O primitives. Section 6 outlines previous work on functional I/O and Section 7 summarises. Appendix A summarises the types of all the I/O and operating system operations provided by Haskell 1.3 and Appendix B contains code for combinator parsing.

2 Elements of Monadic I/O

Monadic I/O depends on the builtin type constructor, `IO`. An expression of some type `IO` *a* denotes a *computation*, that may perform I/O and then returns a result of type *a*. The main program (function `main` from module `Main`, which we write `Main.main`) has type `IO ()`, that is, it is a computation which performs some I/O and returns an uninteresting result. The "trivial" type `()` has only one value, the unit value, which is also written `()`. When a Haskell program runs, there is a single top-level thread of control that executes the computation denoted by `Main.main`. Only this thread of control actually executes the computations denoted by I/O expressions.

The type constructor `IO` is a major extension to Haskell in that it allows many imperative commands to be expressed within a higher-order type-secure language. Unlike languages like Lisp or ML, however, in which arbitrary expressions may have side-effects, only expressions of type `IO` *a* may do so in Haskell, and only then when interpreted as computations by the top-level thread of control. The meaning of expressions is therefore the same as in Haskell 1.2.

Section 2.1 introduces monadic I/O using the handful of I/O operations present in the standard prelude. The majority of operations are in plain control flow and error signalling operations on the `IO` monad in Section 2.3.

2.1 Simple programs

The simplest possible programs just output their result to the standard output device (this will normally be the user's terminal). This is done in Haskell using the `print` function, whose type is given below.

```
print :: Text a => a -> IO ()
```

If *x* has some type *a* which is in the `Text` class, then `print` *x* is the computation that prints `show` *x*, a textual representation of *x*, on the standard output. The `Text` class contains types such as `Int`, `Bool` and `Char`, lists and tuples formed from them, and certain programmer-declared algebraic types. The libraries document [8] defines the `show` function and the `Text` class. Here, for example, is a program to output the first nine natural numbers and their powers of two.

```
main :: IO ()
main =  print [(n, 2^n) | n <- [0..8]]
```

The output of the program is:

```
[(0, 1), (1, 2), (2, 4), (3, 8), (4, 16),
 (5, 32), (6, 64), (7, 128), (8, 256) ]
```

The `show` function, and hence also `print`, formats its output in a standard way, as in source Haskell programs, so strings and characters are quoted (for example, `"Haskell B. Curry"`), special characters are output symbolically (that is, `'\n'` rather than a newline), lists are enclosed in square brackets, and so on. There are other, more primitive functions which can be used to output literal characters or strings without quoting when this is needed (`putChar`, `putStr`). These are described in the following sections.

Interacting with the User

Haskell 1.3 continues to support Landin-stream style interaction with standard input and output, using `interact`. (The type `String` below is a synonym for `[Char]`.)

characters produced to standard output. For example, the following program simply removes all non-upper-case characters from its standard input and echoes the result on its standard output.

```
main = interact (filter isUpper)
```

The functions `filter` and `isUpper` come from the Haskell prelude. They have the following types.

```
filter  :: (a -> Bool) -> [a] -> [a]
isUpper :: Char -> Bool
```

When run on the following input,

```
Now is the time for all Good Men to come to the aid of the Party.
```

this program would output the following.

```
NGMP
```

Since `interact` only blocks on input when demand arises for the lazy input stream, it supports simple interactive programs; see Frost and Launchbury [5], for instance.

Basic File I/O

```
writeFile, appendFile
  :: FilePath -> String -> IO ()
readFile :: FilePath -> IO String
```

The `writeFile` and `appendFile` functions write or append their second argument, a string, to the file named by their first argument. Type `FilePath` is a synonym for `String`. To write a value of any printable type, as with `print`, use the `show` function to convert the value to a string first. For example,

```
main =
  appendFile "ascii-chars"
    (show [(x,chr (x)) | x <- [0..127]])
```

writes the following to the file `ascii-chars`:

```
[(0, '\NUL'), (1, '\SOH'), (2, '\STX'), .
... (126, '~'), (127, '\DEL')]
```

The `readFile` function reads the file named by its argument and returns the contents of the file as a string. The file is read lazily, on demand, as with `interact`.

begins by running *comp*1. When it returns a result x, computation *comp*2 is run, which may depend on x. For example, the following program reads the file `infile`, turns all upper-case characters into lower-case ones, and then writes the result to the file `outfile`.

```
import LibIO

main =
   readFile "infile"  >>= \input ->
     let output = map toLower input
     in
          writeFile "outfile" output
```

The notation `\ p -> e` is a Haskell lambda-expression, denoting a function whose argument is the pattern `p` and whose body is the expression `e`.

This level of programming (treating files as `String`s) was roughly all that could be done with Haskell 1.2, and in fact programs at this simple level can be used almost without change in Haskell 1.3. To write more sophisticated programs than these in Haskell 1.3, one or more I/O libraries need to be explicitly imported. There are seven such libraries: `LibIO`, for basic I/O operations; `LibSystem` for interaction with the system; `LibDirectory` for operations on file directories; `LibUserInterrupt` to handle user interrupts; `LibTime` for operations on clock times; `LibCPUTime` for CPU timing operations; and a library for POSIX-compliant implementations `LibPOSIX`. Most programs will only need to use `LibIO`.

2.2 Character-Based I/O

Stream-based operations, working on complete files or devices, such as `writeFile` or `interact`, are in fact defined in terms of character-based primitives. The two simplest functions are `getChar` and `putChar`.

```
getChar     ::          IO Char
putChar     :: Char -> IO ()
```

The `getChar` computation reads a character c from the standard input device and returns

standard output (equivalently to, but somewhat more verbosely than `interact id`!)

```
import LibIO

main =
   isEOF                    >>= \eof ->
   if eof then return ()
   else
     getChar                >>= \c ->
     putChar c              >>
     main
```

This program uses several new functions. The `return` function simply returns its argument as the result of the monadic computation. The function `(>>)` is identical to `(>>=)` except that its continuation takes no argument: the result, if any, of the first computation is simply discarded. The function `isEOF` returns returns `True` when the end-of-file is reached, and `False` otherwise.

2.3 Results and Errors

I/O operations need to indicate errors without terminating the program, and implementations need to handle these errors. Hence, as well as terminating successfully with a result (for example using `return`), I/O computations may terminate in failure, returning an *error value* of the builtin type `IOError`. For instance, input operations fail with the error value `eofIOError` to indicate end of file. Programmers can generate failures directly via the `fail` function, of type `IOError -> IO ()`. The parsing combinators of Section 5 illustrate `fail`.

So that error values may propagate as intended, the `(>>=)` function needs to take account of the possibility of failure. If the first computation fails with some error value e, then the entire computation also fails with e.

Here is a simple parity checker to compute the parity of an input consisting of just `T`s and `F`s. The function `userError` yields a program-specific error value which is distinct from those generated by the I/O primitives.

```
module Parity where
import LibIO

  case isUserError err of
     Just "Parity" -> True
     _             -> False

parity :: Bool -> IO Bool
parity b =
  isEOF                     >>= \eof ->
  if eof then return b

  else getChar              >>= \c ->
       if c=='T' then
       (if b then parity False
             else parity True)

  else if c=='F' || isSpace c then
    parity b

  else fail parityError
```

The computation `parity True` returns `True` if the number of `T`s is even, and `False` if the number is odd. But if any character other than `T`, `F` or white-space is in the input, the computation fails with the programmer-defined error value `parityError`.

Catching Errors

Failures can be handled by the programmer using the `catch` function, whose type is

```
catch :: IO a -> (IOError->IO a) -> IO a
```

Computation `catch` *comp f* performs computation *comp*. If *comp* returns a result x, this is the result of the entire computation. Otherwise, if *comp* returns an error value x, the computation continues with $f\ x$. For example, the following program handles errors which are detected in the `parity` function, but not those which are generated by the I/O primitives.

```
import LibIO
import LibSystem
import Parity

main = (parity True >>= \p -> print p)
       `catch` handler

handler err =
```

If the `parity` function returns `parityError`, a message is printed and the program is terminated immediately with a failure exit code using the `exitFailure` operation from library `LibSystem` (see Section 4.2. Otherwise, the `handler` function simply propagates the error value using `fail`. If an error value is not caught and handled then the program eventually terminates with a failure code.

There is also an operation `try` which can be used to expose error values in computations that fail, turning the failures into successful computations. The type of `try` is `IO a -> IO (Either IOError a)`, where `Either` is a prelude type defined by the following.

```
data Either a b = Left a | Right b
```

The computation `try` *comp* runs the computation *comp*, and if it returns the successful result x, returns result `Right` x (the "right" answer). Otherwise if *comp* returns an error value x it returns the result `Left` x. Hence `try` *comp* never fails with an error value. Of course it may loop if *comp* loops. The `try` operation can be defined in terms of the `catch` primitive as follows.

```
try p =
  (p >>= (return . Right)) `catch`
  (return . Left)
```

Haskell also defines a type `Maybe` which is similar to `Either`. We will use this type to indicate optional results from functions and computations.

```
data Maybe a = Nothing | Just a
```

For example,

```
isUserError :: IOError -> Maybe String
```

determines whether its argument (an `IOError`) is a programmer-defined error. If so it returns `Just` *err*, where *err* is a programmer-specific string. Otherwise it returns `Nothing`.

The Error Function

Haskell 1.3 continues to support the `error` function. An expression `error` *msg* can be of arbitrary type. It has the same semantics as a Haskell 1.3 as a way of indicating program bugs, for instance, it is better to use error values in computations, since these can be caught and handled appropriately. There is no way to catch an error indicated by the `error` function.

3 The LibIO Library

Having explained the basic operations on the `IO` monad, the objective of this section is to cover the I/O operations provided by the `LibIO` library. We begin in Section 3.1 by defining Haskell files and handles. Section 3.2 explains how files are opened and closed. Section 3.3 explains how to control the buffering of handle I/O and Section 3.4 explains how handles may be repositioned in a file. Operations in Sections 3.5, 3.6 and 3.7 cover querying handle properties, input and output respectively. The types of all these functions are in Appendix A.

3.1 Files and Handles

Haskell interfaces to the external world through an abstract *file system*. This file system is a collection of named *file system objects*, which may be organised in *directories* (see Section 4.1). We call any file system object that isn't a directory a *file*, even though it could actually be a terminal, a disk, a communication channel, or indeed any other object recognised by the operating system. File and directory names are strings. Files can be opened, yielding a handle which can then be used to operate on the contents of that file. Directories can be searched to determine whether they contain a file system object. Files (and normally also directories) can be added to or deleted from directories.

To process files character-by-character, Haskell 1.3 introduces *handles*, which are analogous to ANSI C's file descriptors. A handle is a value of type `Handle` which has at least the following properties:

- whether the handle manages input or output or both;
- whether the handle is *open*, *closed* or *semi-closed* (see Section 3.2);

- a buffer (whose length may be zero).

Most handles will also have a current I/O position indicating where the next input or output operation will occur.

Standard Handles

There are three standard handles which manage the standard input (`stdin`), standard output, (`stdout`), and standard error devices (`stderr`), respectively. The first two are normally connected to the user's keyboard and screen, respectively. The third, `stderr`, is often also connected to the user's screen—a separate handle is provided because it is frequently useful to separate error output from the normal user output which appears on `stdout`. In operating systems which support this separation, one or the other is often directed into a file. If an operating system doesn't distinguish between normal user output and error output, a sensible default is for the two names to refer to the same handle. It is common for the standard error handle to be *unbuffered* (see Section 3.3) so that error output appears immediately on the user's terminal, but this is not always the case.

3.2 Opening and Closing Files

The `openFile` function is used to obtain a new handle for a file.

```
openFile :: FilePath -> IOMode -> IO Handle
```

It takes a *mode* parameter of type `IOMode`, that controls whether the handle can be used for input-only (`ReadMode`), output-only (`WriteMode` or `AppendMode`), or both input and output (`ReadWriteMode`). `ReadWriteMode` allows programmers to make small incremental changes to text files—this can be much more efficient than reading a complete file as a stream and writing this back to a new file. When a file is opened for output, it's created if it doesn't already exist. If, however, the file does exist and it is opened using `WriteMode`, it is first truncated to zero length before any characters are written to it.

on files as follows.

```
import LibIO
import LibSystem

main =
   getArgs                  >>= \args ->
   let (inf:outf:_) = args in
   openFile inf  ReadMode   >>= \ih   ->
   openFile outf WriteMode  >>= \oh   ->
   copyFile ih oh           >>
   hClose ih                >>
   hClose oh

copyFile :: Handle -> Handle -> IO ()
copyFile ih oh =
   hIsEof ih                >>= \eof ->
   if eof then return ()
   else
      hGetChar ih           >>= \c  ->
      hPutChar oh c         >>
      copyFile ih oh
```

The `getArgs` computation (whose type is `IO [String]`) returns a list of strings which are the arguments to the program. The `hClose` function closes a previously opened handle. Once closed, no further I/O can be performed on a handle. In this particular program, the two uses of `hClose` are superfluous, since all open handles are automatically closed when the program terminates. It is generally good practice to close open handles once they are finished with. Many operating systems allow a program only a limited number of live references to file system objects.

Lazy Input Streams

The `hGetContents` function (whose type is `Handle -> IO String`) is used to emulate stream I/O by reading the contents of a handle lazily on demand. For example, the standard `interact` function described earlier can be defined like this:

```
interact f =
   hGetContents stdin     >>= \s ->
   hPutStr stdout (f s)
```

error occurs on a semi-closed handle it is simply discarded. This is because it is not possible to inject error values into the stream of results: `hGetContents` returns a lazy list of characters, and only computations of type `IO` *a* can fail!

Normally semi-closed handles will be closed automatically when the contents of the associated stream have been read completely. Occasionally, however, the programmer may want to force a semi-closed handle to be closed before this happens, by using `hClose` (for instance if an error occurs when reading a handle, or if the entire contents is not needed but the file must be overwritten with a new value). In such a case the contents of the lazy input list are implementation dependent.

File Locking

A frequent problem with Haskell 1.2 was that implementations were not required to lock files when they were opened. Consequently, if a program opened a file again for writing while it was still being read, the results returned from the read could be garbled. Because of lazy evaluation and implicit buffering (also not specified by Haskell 1.2), it was possible for this to happen on some but not all program executions. This problem only occurs with languages which implement lazy stream input (à la `hGetContents`) and also have non-strict semantics.

In general it is hard for programmers to avoid opening a file when it has already been opened in an incompatible way. Almost all non-trivial programs open user-supplied filenames, and there is often no way of telling from the names whether two filenames refer to the same file. The only safe thing to do is implement file locks whenever a file is opened. This could be done by the programmer if a suitable locking operation was provided, but to be secure such locking would need to be done on every `openFile` operation, and might also require knowledge of the operating system.

The definition of Haskell 1.3 therefore requires that identical files are locked against accidental overwriting within a single Haskell program (single-writer, multiple-reader). Two physical user's data files. Even so, the definition *only* requires an implementation to take precautions to avoid obvious and persistent problems due to lazy file I/O (a language feature): it *does not* require the implementation to protect against interference by other applications or the operating system itself.

File Size and Extent

For a handle *hdl* which attached to a physical file, computation `hFileSize` *hdl* returns the total size of that file as an integral number of bytes.

```
hFileSize :: Handle -> IO Integer
```

On some operating systems it is possible that this will not be an accurate indication of the number of characters that can be read from the file.

On some systems, such as the Macintosh, it is much more efficient to define the maximum size of a file (or *extent*) when it is created, and to modify this extent if the file changes. This may allow a file to be laid out contiguously on disk, for example, and therefore accessed more efficiently. In any case, the actual file size will be no greater than the extent. While efficient file access is a desirable characteristic, we felt that dealing with file extents was over-complex for the normal programmer.

3.3 Buffering

Explicit control of buffering is important in many applications, including ones that need to deal with raw devices (such as disks), ones which need instantaneous input from the user, or ones which are involved in communication. Examples might be interactive multimedia applications, or programs such as `telnet`. In the absence of such strict buffering semantics, it can also be difficult to reason (even informally) about the contents of a file following a series of interacting I/O operations.

Three kinds of buffering are supported by Haskell 1.3: line-buffering, block-buffering or no-buffering. These modes have the following

overflows, a flush is issued, or the handle is closed.

- **block-buffering:** the entire buffer is written out whenever it overflows, a flush is issued, or the handle is closed.

- **no-buffering:** output is written immediately, and never stored in the buffer.

The buffer is emptied as soon as it has been written out.

Similarly, input occurs according to the buffer mode for handle *hdl*.

- **line-buffering:** when the buffer for *hdl* is not empty, the next item is obtained from the buffer; otherwise, when the buffer is empty, characters up to and including the next newline character are read into the buffer. No characters are available until the newline character is available.

- **block-buffering:** when the buffer for *hdl* becomes empty, the next block of data is read into the buffer.

- **no-buffering:** the next input item is read and returned.

For most implementations, physical files will normally be block-buffered and terminals will normally be line-buffered.

The computation `hSetBuffering` *hdl mode* (whose type is `Handle -> BufferMode -> IO ()`) sets the mode of buffering for handle *hdl* on subsequent reads and writes as follows.

- If *mode* is `LineBuffering`, then line-buffering is enabled if possible.

- If *mode* is `BlockBuffering` *m*, then block-buffering is enabled if possible. The size of the buffer is *n* items if *m* is `Just` *n* and is otherwise implementation-dependent.

- If *mode* is `NoBuffering`, then buffering is disabled if possible.

If the mode is changed from `BlockBuffering` or `LineBuffering` to `NoBuffering`, then

The default buffering mode when a handle is opened is implementation-dependent and may depend on the object which is attached to that handle. The three buffer modes mirror those provided by ANSI C.

Flushing Buffers

Sometimes implicit buffering is inadequate, and buffers must be flushed explicitly. The computation `hFlush` *hdl* (whose type is `Handle -> IO ()`) causes any items buffered for output in handle *hdl* to be sent immediately to the operating system. While it would, in principle, be sufficient to provide only `hFlush` and so avoid the complexity of explicit buffer setting, this would be tedious to use for any kind of buffering other than `BlockBuffering`, and would make it harder to write library functions that worked for different kinds of buffering.

3.4 Re-positioning Handles

Many applications need direct access to files if they are to be implemented efficiently. Examples are text editors, or database applications. These applications often work on read-write handles. The design given here draws heavily on the ANSI C standard.

Seeking to a new I/O position

Many operating systems, including Unix and the Macintosh, allow I/O at any position in a file. The `hSeek` operation allows three kinds of file positioning: absolute positioning `AbsoluteSeek`, positioning relative to the current I/O position `RelativeSeek`, and positioning relative to the current end-of-file `SeekFromEnd`. For simplicity, all positioning offsets are an integral number of bytes.

```
hSeek :: Handle -> SeekMode -> Integer
                            -> IO ()
```

Revisiting an I/O position

On some operating systems or devices, it is not possible to seek to arbitrary locations, but only

to that point. Absolute seeking is not sensible in this case. Functions `hGetPosn` and `hSetPosn` together provide the ability to revisit a previously visited file position, using an abstract type to represent the positioning information. To improve portability, there is no standard way to convert a `handlePosn` into an `Integer` offset or to compare different file positions.

```
hGetPosn :: Handle     -> IO HandlePosn
hSetPosn :: HandlePosn -> IO ()
```

For example, if both `hSeek` and `hGetPosn` are supported, then the following function could be written to append a string to a file, and return the position where it was appended.

```
module Append where
import LibIO

append :: Handle -> String
                 -> IO HandlePosn
append h s =
   hSeek h SeekFromEnd 0   >>
   hGetPosn h              >>= \pos ->
   hPutStr h s
   return pos
```

3.5 Handle Properties

There are several functions that query a handle to determine its properties: `hIsOpen`, `hIsClosed`, `hIsReadable`, `hIsSeekable` and so on. These all have type `Handle -> IO Bool`. Originally we considered a single operation to return all the properties of a handle. This proved to be very unwieldy, and would also have been difficult to extend to cover other properties (since Haskell does not have named records). The operation was therefore split into many component operations, one for each property that a handle must have. Determining the current I/O position is treated as a separate operation.

While there are `hIsOpen` and `hIsClosed` operations, there is no way to test whether a handle is semi-closed. This was felt to be of marginal utility for most programmers, and is easy to define if necessary.

```
   hIsOpen h               >>= \ho ->
   hIsClosed               >>= \hc ->
   return (not (ho || hc))
```

3.6 Text Input

The function `hReady` determines whether input is available on a handle. It is intended for writing interactive programs or ones which manage multiple input streams. Because `hReady` is non-blocking, beware that this could be extremely inefficient if it is executed too frequently. The function `hLookAhead` can be used to inspect the next input character without removing it from the buffer. This is useful when writing programs such as lexical analysers that need to look ahead in the input stream.

```
hReady     :: Handle -> IO Bool
hLookAhead :: Handle -> IO Char
```

3.7 Text Output

Most of the text output operations which are provided have already been described. The distinction between `hPutStr` and `hPutText` is worth emphasising, however.

```
hPutStr  :: Handle -> String -> IO ()
hPutText :: Text a => Handle -> a
                              -> IO ()
```

Function `hPutText` outputs any value whose type is an instance of the `Text` class, quoting strings and characters as necessary. Function `hPutStr`, on the other hand, outputs an unformatted stream of characters, so that tabs appear as literal tab characters in the output and so on. For example, the following outputs the two words `Hello` and `World` on a line, separated by a tab character,

```
import LibIO
main = putStr stdout "Hello\tWorld\n"
```

whereas the following outputs the string `"Hello\tWorld\n"`.

```
import LibIO
main = putText stdout "Hello\tWorld\n"
```

- retrieve the current working directory (`getCurrentDirectory`);
- set the current directory to a new directory (`setCurrentDirectory`);
- list the contents of a directory (`getDirectoryContents`);
- delete files or directories (`removeFile` and `removeDirectory`);
- and to rename files or directories (`renameFile` and `renameDirectory`).

```
getCurrentDirectory  :: IO FilePath
getDirectoryContents
  :: FilePath -> IO [FilePath]
removeDirectory, removeFile
  :: FilePath -> IO ()
renameDirectory, renameFile
  :: FilePath -> FilePath -> IO ()
```

4.2 LibSystem

The `LibSystem` library defines a set of functions which are used to interact directly with the Haskell program's environment. The most important of these are `system`, which introduces a new operating system process and waits for the result of that process, and `getArgs` which returns the command-line arguments to the program.

```
system  :: String -> IO ExitCode
getArgs :: IO [String]
```

It is possible that neither of these functions is available on a particular system; for example, these commands do not generally make sense under the Macintosh operating system. When using `system` note that the commands which are produced are operating system dependent. It is entirely possible that these commands may not be available on someone else's system, so programs which use `system` may not be portable. Here is how to create a soft-linked alias to a file under Berkeley or similar Unixes.

```
  system ("ln -s "++old++" "++new)
```

Exit Codes

As described earlier, programs can terminate immediately and return an exit code to the operating system. In general this is done using the `exitWith` operation.

```
exitFailure :: IO a
exitWith    :: ExitCode -> IO a
```

The argument to `exitWith` is of type `ExitCode`, whose only constructors are `ExitSuccess` and `ExitFailure`. Haskell 1.3 assumes that the operating system understands numeric return codes. Function `exitWith` maps `ExitFailure` *exitfail* to a computation that immediately terminates the Haskell program and sends the operating system the numeric code *exitfail*. Likewise, `exitWith ExitSuccess` immediately terminates Haskell and sends the code for success, the number being dependent on the operating system.

Environment Variables

Simple access to environment variables is supported through the `getEnv` computation, whose type is `String -> IO String`. Environment variables are supported by many operating systems, and provide a useful way of communicating infrequently-changed information to a program. When available, the use of environment variables can significantly reduce the length of textual command lines, or the options which must be set in graphical user dialogues.

4.3 LibTime and LibCPUTime

The `LibTime` library provides operations that access time and date information (useful for timestamping or for timing purposes), including simple date arithmetic and simple text output. It codifies existing practice in the shape of the `Time` library provided by `hbc`. Unlike that library it is not Unix-specific, and it provides support for international time standards, including time-zone information. Time differences are

```
diffClockTimes
 :: ClockTime -> ClockTime -> TimeDiff

toCalendarTime, toUTCTime
  :: ClockTime -> CalendarTime

toClockTime :: CalendarTime -> ClockTime
```

The `LibCPUTime` library defines exactly one function to access the total CPU time that a program has used to date, `getCPUTime` of type `IO Integer`.

4.4 LibUserInterrupt

User-produced interrupts are the most important class of interrupt which programmers commonly want to handle. Almost all platforms, including small systems such as Macintosh and MS/DOS, provide some ability to generate user-produced interrupts.

User interrupts can be handled in Haskell if a handler is installed using `setUserInterrupt`.

```
setUserInterrupt :: Maybe (IO ())
                 -> IO (Maybe (IO ()))
```

Whenever a user interrupt occurs, the program is stopped. If an interrupt handler is installed, this is then executed in place of the program. If no interrupt handler is installed, the program is simply terminated with an operating system failure code. For example, the following program installs an interrupt handler `ihandler` that prints `^C` on `stdout` and then continues with some new computation.

```
import LibUserInterrupt

main = setUserInterrupt ihandler >>
       ...
ihandler = (putStr "^C") >> ...
```

4.5 LibPOSIX

A library (`LibPOSIX`) has been defined that builds on the basic monadic I/O definition to provide a complete interface to POSIX-compliant operating systems. There is insufficient space to describe this library in detail here,

5 Combinator Parsing

In this section we illustrate monadic I/O in Haskell by writing a lexer and parser for untyped lambda-calculus. Our parser recognises strings of characters input from a handle. The characters are first grouped into *tokens* by the lexer. The parser acts on the sequence of tokens.

A Lexer

A token is either an alphanumeric identifier (beginning with a letter), a special symbol from the following list,

```
symbols = "()\\="
```

or else an illegal character. Tokens are represented by the following datatype.

```
data Token = ALPHA String | SYMBOL Char
           | ILLEGAL Char | EoF
           deriving (Eq, Text)
```

The `EoF` token indicates end of file. Here is a simple lexer.

```
hGetToken :: Handle -> IO Token
hGetToken h =
  hIsEOF h                  >>= \eof ->
  if eof then return EoF
  else
     hGetChar h             >>= \c ->
     if isSpace c then hGetToken h else
     if isAlpha c then hGetAlpha h [c]
     else if c `elem` symbols then
        return (SYMBOL c)
     else
        return (ILLEGAL c)

hGetAlpha :: Handle -> String
                    -> IO Token
hGetAlpha h cs =
  hIsEOF h                  >>= \eof ->
  if eof then
     return (ALPHA (reverse cs))
  else
     hLookAhead h           >>= \c ->
     if isAlphanum c then
       hGetChar h >> hGetAlpha h (c:cs)
    else
       return (ALPHA (reverse cs))
```

We can write predictive recursive-descent parsers [2] using combinators. In a predictive parser the lookahead token unambiguously determines the recursive function to be applied at each point.

Our type of parsers is a parameterised state-transformer monad built from the `IO` monad.

```
type Parser a =
  Handle -> Token -> IO (a, Token)
```

Given a handle h and a lookahead token $tok0$, a parser of type `Parser` a may do one of three things.

Accept a phrase with result x :: a.
The parser consumes the tokens of the phrase using `hGetToken` h and then returns $(x,tok1)$ where $tok1$ is the new lookahead token.

Fail with a lookahead error.
The parser consumes no tokens and immediately fails with an error value of the form `UserError ('L':`msg`)`, a *lookahead error*.

Fail with a parse error. The
parser consumes zero or more tokens and then fails with an error value of the form `UserError ('P':`msg`)`, a *parse error*.

Failure with a lookahead error is used to select alternatives based on the lookahead token; failure with a parse error indicates an unparsable input. The difference between parse and lookahead errors is coded using the first character of the error string. It would be better to use two different constructors, but there is no way for programs to extend `IOError`.

The top of Appendix B shows operations on error values. Computation `lookaheadError` x y immediately fails with a lookahead error indicating that x was expected by y was found. Predicate `isLookahead` determines whether an error value is a lookahead error. Computation `mkParseError` e turns lookahead errors into parse errors.

The middle of Appendix B shows the implementation of the `Parser` monad. Token matching is performed by `match`. Its second argument is a predicate of type `Token -> Maybe` a. the parser's result is y. Otherwise if the outcome is `Nothing`, meaning that the lookahead is rejected, the parser immediately fails with a lookahead error.

If p and q are parsers, p `` `alt` `` q is the parser that accepts all the phrases accepted by either p or q, provided that the choice is determined by the lookahead token. The parser first runs parser p. If p either accepts a phrase or fails with a parse error, then so does p `` `alt` `` q. But if p fails with a lookahead error—in which case the lookahead is unchanged but rejected—then q is run instead.

Functions `returnP` and `thenP` are the two standard monadic functions, analogous to `return` and `>>=` on the `IO` monad. Parser `returnP` x accepts the empty phrase and returns result x. If parser p accepts a phrase with result x, then p `` `thenP` `` f consumes that phrase and then acts as parser $f(x)$. Any lookahead error from $f(x)$ must be turned into a parse error because p may already have consumed tokens. If parser p fails with a lookahead or parse error, then so does p `` `thenP` `` f.

Finally, if p is a parser and h a handle, `parse` p is the computation that runs p on the tokens obtainable using `hGetToken` h.

The primitives in Appendix B are enough to build arbitrary predictive parsers. The bottom of the appendix shows some derived parser functions. Parser `theToken` tok accepts the token tok and returns it as its result. Parser `ident` accepts any alphanumeric token, and returns its `String` representation. On any other input, both these parsers fail with a lookahead error.

Function `seqP` is an unparameterised form of `thenP`, analogous to `>>`. Function `><` runs two parsers in sequence, and returns their results as a pair. If p is a parser, `repeatP` p applies p repeatedly until it fails with a lookahead error; it returns the list of accepted results as its result.

A Parser

Suppose we want to parse untyped lambda-calculus programs such as the following.

```
true =  \(x)\(y)x
false = \(x)\(y)y
```

```
decl  = {ident "=" exp} EOF
exp   = ident
      | "\" "(" ident ")" exp
      | exp "(" exp ")"
```

The conventions are that $X\ Y$ means X followed by Y, X | Y means X or Y, and $\{X\}$ means a possibly empty sequence of X's. The following datatype represents lambda-terms.

```
data Exp = VAR String
         | LAM String Exp
         | APP Exp Exp
         deriving Text
```

As usual, we must remove left-recursion to make the grammar suitable for recursive descent parsing.

```
decl0  = {decl1} EoF
decl1  = ident "=" exp0
exp0   = exp1 { exp2 }
exp1   = ident | "\" "(" ident ")" exp0
exp2   = "(" exp0 ")"
```

The following recursion equations represent this transformed grammar as predictive parsers.

```
decl0 =
  repeatP decl1 `thenP` \x ->
  theToken EoF  `seqP`  returnP x

decl1 =
  ident `thenP` \x -> eq `seqP`
  exp0  `thenP` \t -> returnP (x,t)

exp0 =
  exp1          `thenP` \t ->
  repeatP exp2 `thenP` \ts ->
  returnP (foldl APP t ts)

exp1 =
  (ident `thenP` (returnP . VAR))
  `altP`
  (lambda `seqP`  lp       `seqP`
   ident  `thenP` \x -> rp `seqP`
   exp0   `thenP` \t ->
    returnP (LAM x t))

exp2 =
  map (theToken . SYMBOL) symbols
```

If our main program is

```
main :: IO ()
main = parse decl0 stdin >>= \p ->
       print p
```

here is its output on the declarations shown at the beginning of this section.

```
[("true",  LAM "x" (LAM "y" (VAR "x"))),
 ("false", LAM "x" (LAM "y" (VAR "y"))),
 ("cond",  LAM "b" (LAM "t" ...)),
 ("zero",  LAM "f" (LAM "x" ...)),
 ("succ",  LAM "n" (LAM "f" ...))]
```

Discussion

Combinator parsers—like any other recursive descent parsers—are less efficient than bottom-up table-driven parsers. But they can be quickly and simply written, and for many purposes they are fast enough. Some previous parsers represented their input as a list, and hence supported arbitrary lookahead [15]. In comparison, our parsers manage their input directly using `hGetToken` and are predictive—they use only a single lookahead token. Managing arbitrary lookahead would require significant reorganisation of the program. Other parsers represented their output as a list of possible parses, to cater for ambiguous grammars [3, 5, 12]. While our parsers only return a single successful parse, this is sufficient for many computer languages.

Of course, Haskell 1.3 continues to support stream-style parsing via the `interact` function. The standard prelude includes simple parsers of type

```
type ReadS a = String -> [(a,String)]
```

and pretty-printers for types in the `Text` class. Our monad is more flexible than the `ReadS` style because it allows parsing to be freely mixed with other computations.

Chapter 9 of Paulson's book [15] is a good starting point.

(2) Extend the lexer to recognise numerals. Extend the grammar and parser with syntax for numerals and binary arithmetic operators.

(3) Rewrite the lexer using a Haskell array to dispatch on whether the next character is whitespace, alphabetic, symbolic or illegal.

(4) Find a grammar that can be parsed with arbitrary lookahead but not by a predictive parser.

(5) Modify the `Parser` monad to admit arbitrary lookahead. Hint: use the following definition of `Parser`, which explicitly represents lookahead errors rather than using the builtin error-handling mechanism.

```
type Parser a =
  Handle -> [Token] ->
  IO ([Token], Maybe (a, [Token]))
```

If such a parser is run on a handle h with lookahead *toks*, it returns pair (*toks1*, m) where *toks1* is the new lookahead, and m is either `Nothing` if the parse has failed or `Just` (*x*,*toks2*) if the parse was successful. In the latter case, x is the result of the parse and *toks2* is the list of tokens accepted.

6 History and Related Work

In 1989, Cupitt [4] built a functional operational system (KAOS) in Miranda. He was the first to make large-scale use of types, similar to `IO` a, for computations returning an answer of type a. His work also uses a sequential composition operator, similar to `(>>=)`. Independently, about the same time, Gordon [6] proposed a concurrent language called PFL+ with similar constructs. 1989 was also the year Moggi first published his theory of modular denotational semantics [14] based on the categorical notion of a

I/O in general, and monadic I/O in particular, is surveyed in Gordon's book [7]. The contribution of Haskell 1.3 is a detailed standard for portable monadic I/O in Haskell, using handles to access the file system.

Returning an error value from a computation is analogous to raising an exception in a language like Standard ML, except that in Haskell only expressions of `IO` type may return an error value. Hammond's book [10] discusses the use of error values in functional languages.

6.1 Computations and Effects

The type `IO` a denotes computations in the same sense as `Integer` denotes integers and `Bool` denotes truth-values. To a first approximation, we can think of computations as functions which take the state of the world as their argument and return a pair of an updated world and a result [17]. The main thread, defined by `Main.main`, is a sequence of state-transforming computations of type `IO` a, which directly express effects on the environment, such as character I/O, or reading and writing files. Each of the sequence of computations is applied to an implicit program state, to produce a new state together with an intermediate result. The new state and result is passed to the next computation in the sequence, and so on until the program terminates.

Within the Haskell program, expressions of type `IO` a behave identically to other expressions: they may appear evaluated or unevaluated in lists, be freely copied, and so on. Haskell expressions do not have side-effects unless they are evaluated by the top-level thread of control.

6.2 Parallelism

The interaction of I/O with parallelism is important, especially for extensions of Haskell such as the `pH` language. Handled carelessly, I/O could unnecessarily serialise computations and thus reduce performance. Some thought has gone into this. The semantics of I/O is serialisable in the sense that I/O operations interact with the operating system in the order they are pre-

gated as defined by the serial semantics. This may require a mechanism similar to that needed for controlling other speculative computations.

7 Summary

We have presented a design for I/O which has been adopted in the Haskell standard, describing some interesting aspects of the design and providing a tutorial on how it can be used effectively. Being based on the use of monads, the design is both flexible and extensible. Although only a fairly conservative basic design has been provided initially, we expect this to form the basis for more radical research departures later, such as standard libraries for graphical interaction. It already provides much useful functionality that was not previously available in Haskell 1.2.

No formal semantics for these I/O primitives is possible at present, because there is no complete formal semantics for Haskell itself. We hope in future that such a semantics will be developed. One task of such a semantics would be to show that the `IO` type does indeed form a monad in the categorical sense.

Haskell 1.3 allows programmers to write programs that can change the external or global states in an imperative fashion, but only via expressions of some type `IO` *a*, and only when they are then interpreted by the top-level thread of control. This contrasts with languages like LISP or ML, where expressions of any type can have side-effects. Our hope is that I/O in Haskell 1.3 will be no less expressive than in these languages, and that its type system can be exploited by programmers and compilers to yield clear and efficient programs.

Acknowledgements

We are grateful to the other members of Haskell committee who have made many constructive comments on the I/O design during its period of incubation. We would also like to thank those people who have either contributed directly to the I/O design, or whose comments have had dra Loosemoore, and Alastair Reid. We are also grateful to Will Partain and Hans Loidl for commenting on draft versions of this paper and to the anonymous referees who reviewed this paper. We are also grateful to our sponsors. Gordon began this work while a member of the Programming Methodology Group at Chalmers University. At Cambridge, he was supported by the TYPES BRA and a University Research Fellowship from the Royal Society. Hammond was supported at Glasgow by an SOED personal Research Fellowship from the Royal Society of Edinburgh.

References

[1] P. Achten and M. J. Plasmeijer. The ins and outs of Concurrent Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] W. H. Burge. *Recursive Programming Techniques*. Addison Wesley, 1975.

[4] J. Cupitt. A brief walk through KAOS. Technical Report 58, Computing Laboratory, University of Kent at Canterbury, February 1989.

[5] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2):108–121, April 1989.

[6] A. D. Gordon. PFL+ : A kernel scheme for functional I/O. Technical Report 160, University of Cambridge Computer Laboratory, February 1989.

[7] A. D. Gordon. *Functional Programming and Input/Output*. Cambridge University Press, 1994.

[8] K. Hammond et al. Report on the functional programming language Haskell: Version 1.3. Yale University Technical Report, to appear, June 1995.

[10] K. Hammond. *PSML: A Functional Language and its Implementation in Dactl*. Pitman Press, 1991.

[11] P. Hudak, S.L. Peyton Jones, P. L. Wadler, et al. Report on the functional programming language Haskell: Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[12] G. J. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[13] K. Karlsson. Nebula: A functional operating system. Programming Methodology Group, Chalmers University of Technology and University of Gothenburg, 1981.

[14] E. Moggi. Computational lambda calculus and monads. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, June 1989.

[15] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[16] N. Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Department of Computing, Imperial College, London, June 1991.

[17] S. L. Peyton Jones and P. L. Wadler. Imperative functional programming. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 71–84. ACM Press, 1993.

[18] P. L. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*. ACM Press, 1992.

```
infixr 1 >>, >>=                                                        -- Prelude

type IO a                                                               -- Prelude

type Handle                                                             -- LibIO

type FilePath = String                                                  -- LibIO

data IOMode     = ReadMode | WriteMode | AppendMode | ReadWriteMode     -- LibIO

data BufferMode = NoBuffering | LineBuffering | BlockBuffering (Maybe Int)  -- LibIO

type HandlePosn                                                         -- LibIO
data SeekMode   = AbsoluteSeek | RelativeSeek | SeekFromEnd             -- LibIO

data ExitCode   = ExitSuccess | ExitFailure Int                         -- LibSystem

data ClockTime                                                          -- LibTime
instance Ord ClockTime                                                  -- LibTime
instance Eq  ClockTime                                                  -- LibTime
instance Text ClockTime                                                 -- LibTime

data CalendarTime = CalendarTime  Int  Int  Int  Int                    -- LibTime
                                  Int  Int  Integer
                                  Int  Int  String
                                  Int Bool

data TimeDiff    =  TimeDiff Int  Int                                   -- LibTime
                             Int  Int  Int  Int  Integer
                             deriving (Eq,Ord)

stdin, stdout, stderr :: Handle                                         -- LibIO
```

Operations

The set of I/O operations is ordered alphabetically.

```
(>>)                 :: IO a     -> IO b               -> IO b                 -- LibIO
(>>=)                :: IO a     -> (a -> IO b)        -> IO b                 -- LibIO
accumulate           :: [IO a]                         -> IO [a]               -- LibIO
addToClockTime       :: TimeDiff  -> ClockTime         -> ClockTime            -- LibTime
appendFile           :: FilePath -> String             -> IO ()                -- Prelude
catch                :: IO a     -> (IOError -> IO a)  -> IO a                 -- LibIO
createDirectory      :: FilePath                       -> IO ()                -- LibDirectory
diffClockTimes       :: ClockTime -> ClockTime         -> TimeDiff             -- LibTime
exitFailure          ::                                   IO a                 -- LibSystem
exitWith             :: ExitCode                       -> IO a                 -- LibSystem
fail                 :: IOError                        -> IO a                 -- LibIO
getArgs              ::                                   IO [String]          -- LibSystem
getChar              ::                                   IO Char              -- LibIO
getClockTime         ::                                   IO ClockTime         -- LibTime
getCPUTime           ::                                   IO Integer           -- LibCPUTime
getCurrentDirectory  ::                                   IO FilePath          -- LibDirectory
getDirectoryContents :: FilePath                       -> IO [FilePath]        -- LibDirectory
getEnv               :: String                         -> IO String            -- LibSystem
getProgName          ::                                   IO String            -- LibSystem
hClose               :: Handle                         -> IO ()                -- LibIO
hFileSize            :: Handle                         -> IO Integer           -- LibIO
hFlush               :: Handle                         -> IO ()                -- LibIO
hGetBuffering        :: Handle                         -> IO (Maybe BufferMode) -- LibIO
```

```
hIsReadable           :: Handle                          -> IO Bool               -- LibIO
hIsSeekable           :: Handle                          -> IO Bool               -- LibIO
hIsWritable           :: Handle                          -> IO Bool               -- LibIO
hLookAhead            :: Handle                          -> IO Char               -- LibIO
hPutChar              :: Handle   -> Char                -> IO ()                 -- LibIO
hPutStr               :: Handle   -> String              -> IO ()                 -- LibIO
hPutText              :: Text a   => Handle     -> a     -> IO ()                 -- LibIO
hReady                :: Handle                          -> IO Bool               -- LibIO
hSeek                 :: Handle   -> SeekMode -> Integer -> IO ()                 -- LibIO
hSetBuffering         :: Handle   -> BufferMode          -> IO ()                 -- LibIO
hSetPosn              :: HandlePosn                      -> IO ()                 -- LibIO
interact              :: (String -> String)              -> IO ()                 -- Prelude
ioeGetFileName        :: IOError                         -> Maybe FilePath        -- LibIO
ioeGetHandle          :: IOError                         -> Maybe Handle          -- LibIO
isAlreadyExistsError  :: IOError                         -> Bool                  -- LibIO
isAlreadyInUseError   :: IOError                         -> Bool                  -- LibIO
isEOF                 ::                                    IO Bool               -- LibIO
isEOFError            :: IOError                         -> Bool                  -- LibIO
isFullError           :: IOError                         -> Bool                  -- LibIO
isIllegalOperation    :: IOError                         -> Bool                  -- LibIO
isPermissionError     :: IOError                         -> Bool                  -- LibIO
isUserError           :: IOError                         -> Maybe String          -- LibIO
openFile              :: FilePath -> IOMode              -> IO Handle             -- LibIO
print                 :: Text a =>   a                   -> IO ()                 -- Prelude
putChar               :: Char                            -> IO ()                 -- LibIO
putStr                :: String                          -> IO ()                 -- LibIO
putText               :: Text a   => a                   -> IO ()                 -- LibIO
readFile              :: FilePath                        -> IO String             -- Prelude
removeDirectory       :: FilePath                        -> IO ()                 -- LibDirectory
removeFile            :: FilePath                        -> IO ()                 -- LibDirectory
renameDirectory       :: FilePath -> FilePath            -> IO ()                 -- LibDirectory
renameFile            :: FilePath -> FilePath            -> IO ()                 -- LibDirectory
return                :: a                               -> IO a                  -- LibIO
sequence              :: [IO a]                          -> IO ()                 -- LibIO
setCurrentDirectory   :: FilePath                        -> IO ()                 -- LibDirectory
setUserInterrupt      :: Maybe (IO ())                   -> IO (Maybe (IO ()))    -- LibUserInterrupt
system                :: String                          -> IO ExitCode           -- LibSystem
toCalendarTime        :: ClockTime                       -> CalendarTime          -- LibTime
toClockTime           :: CalendarTime                    -> ClockTime             -- LibTime
toUTCTime             :: ClockTime                       -> CalendarTime          -- LibTime
try                   :: IO a                            -> IO (Either IOError a) -- LibIO
userError             :: String                          -> IOError               -- LibIO
writeFile             :: FilePath -> String              -> IO ()                 -- Prelude
```

```
lookaheadError :: String -> String -> IO a
isLookahead    :: IOError -> Bool
mkParseError   :: IOError -> IO a

lookaheadError exp fnd = fail (userError ("L: Expected "++exp++" but found "++fnd))

isLookahead e = case (isUserError e) of {Just ('L':_) -> True; _ -> False}

mkParseError e = case (isUserError e) of
  Just ('L':msg) -> fail (userError ('P':msg))
  _ -> fail e

-- ============================================================================
-- Implementation of the Parser Monad
-- ============================================================================

match   :: String -> (Token -> Maybe a) -> Parser a
altP    :: Parser a -> Parser a -> Parser a
returnP :: a -> Parser a
thenP   :: Parser a -> (a -> Parser b) -> Parser b
parse   :: Parser a -> Handle -> IO a

match e f h tok0 =
  case (f tok0) of
     Just x  -> hGetToken h >>= \tok1 -> return (x, tok1)
     Nothing -> lookaheadError e (show tok0)

p1 `altP` p2 = \ h s ->  p1 h s `catch` \e ->
                         if isLookahead e then p2 h s else mkParseError e

returnP a h s =          return (a, s)
p `thenP` f   = \ h s -> p h s >>= \(a,s) -> (f a h s `catch` mkParseError)
parse p h     =          (hGetToken h >>= p h) >>= (return . fst)

-- ============================================================================
-- Derived Parser Functions
-- ============================================================================

theToken     :: Token -> Parser Token
ident        :: Parser String
seqP         :: Parser a -> Parser b -> Parser b
(><)         :: Parser a -> Parser b -> Parser (a,b)
repeatP      :: Parser a -> Parser [a]

theToken tok =  match (show tok) (\tok0 -> if tok==tok0 then Just tok else Nothing)
ident        =  match "<ident>"
                      (\tok0 -> case tok0 of {ALPHA x -> Just x; _ -> Nothing})
p1 `seqP` p2 =  p1 `thenP` const p2
p1 >< p2     =  p1 `thenP` \x -> p2 `thenP` \y -> returnP (x,y)
repeatP p    =  (p >< repeatP p `thenP` (returnP . uncurry (:))) `altP` returnP []
```