# Appendix D

# Lexical analysis, parsing, and reading input using ML

Contents



SUMMARY TABLE OF ALL COMBINATORS!

How is a program represented? If you have worked through this book, you will believe (I hope) that the most fundamental and most useful representation of a program is its abstract-syntax tree. But syntax trees aren't easy to create or specify directly, so unless they have access to a special-purpose language-based editor (perhaps as part of an integrated development environment), programmers have to specify an abstract-syntax tree indirectly by writing a sequence of characters. The process of turning a sequence of characters into syntax is called *parsing*.

Wait! It gets better. Quite often characters are turned into syntax in *two* stages: first characters are grouped together into *tokens*. Then, a parser turns a sequence of tokens into syntax. Think of a token as a word or a symbol or a piece of punctuation.

Parsing is a deep and broad topic with many intellectual byways. As long ago as 1970 it was already possible to devote a 500-page monograph to parsing, and since then, clever minds have invented plenty of new techniques. Many techniques rely on a separate tool called a *parser generator*. The technique I use in this book requires no separate tools: I use *hand-written, recursive-descent parsers*. To help me write parsers by hand, I have created[1] a set of higher-order functions designed especially to manipulate parsers. Such functions are known as *parsing combinators*. My parsing combinators appear in this appendix.

Most parsing techniques have been invented for use in compilers. and a typical compiler swallows programs in large gulps, one file at a time. Unlike these typical compilers, the interpreters in this book are interactive, and they swallow just one *line* at a time. Interactivity imposes additional requirements:

- The parser might cooperate with the I/O routines to arrange that a suitable *prompt* is issued before each line is read. the prompt should tell the user whether the parser is waiting for a new definition or is in the middle of parsing a current definition.
- If the parser encounters an error, it can't just give up. It needs get itself back into a state where the user can continue to interact.

These requirements make my parsing combinators a bit different from standard ones. In particular, in order to be sure that the actions of printing a prompt and reading a line of input occur in the proper sequence, I manage these actions using an abstraction called *lazy streams*. Unlike the lazy streams built into Haskell, these lazy streams can do input and output and can perform other actions.

This appendix presents all the abstractions I use to build interactive parsers:

- Lazy streams, which are themselves based on *suspensions*
- The *error monad*, which tracks parsing errors and enables parsers to recover
- Parsing combinators, which help turn sequences into tokens or syntax
- A *reader*, which ties everything together.

These abstractions are coded in a number of different parts:

644 ⟨*support for streams, lexical analysis, and parsing* 644⟩≡ (725a)
⟨*suspensions* 646⟩
⟨*streams* 647a⟩
⟨*error handling* 651a⟩
⟨*parsing combinators* 652c⟩
⟨*support for lexical analysis* 659a⟩
⟨*support for parsing* 660d⟩
⟨*parsing utilities* 662b⟩
⟨*code used to debug parsers* 666b⟩
⟨*an interactive reader* 667b⟩

The abstractions are useful for reading all kinds of input, not just computer programs, and I encourage you to use them in your own projects. But here are two words of caution: with so many abstractions in the mix, the parsers are tricky to debug. And while some parsers built from combinators are very efficient, mine aren't.

[1] I say "created," but a more accurate term would be "stolen."

## D.1 Controlling actions by using lazy streams

The simplest way to write a parser by hand, say for a language like μScheme, is to try alternatives. We can ask

- Is the next line of input a `val`?
- Is the next line of input an expression?
- Is the next line of input a `define`?
- Is the next line of input a `use`?

To ask about input, I have to *read* input. But if I read a line of input each time I ask a question, there's a potential disaster: suppose you type a line like `(+ 1 2)`, and I read that line, discover that it isn't a `val`, so I then ask if the next line is an expression, and to do so *I read another line?* Now I've just thrown away your input. Not good.

Like most programming languages, ML treats input (and output) as an imperative action. An action like reading a line of input, once performed, can never be repeated. To be able to look at the same line multiple times, I use an abstraction that is more powerful and flexible than a "current line" that could be lost when the next line is read. That abstraction is a *lazy stream*, and it records the result of a *sequence* of actions. By hiding the action of reading behind the stream abstraction, we can treat an input as an immutable sequence of lines... or characters... or definitions. The stream puts ephemeral results of unrepeatable actions into a data structure that we can hold into as long as we like and examine as many times as we like.

### D.1.1 Suspensions: repeatable access to the result of one action

The building block of lazy computation is the *suspension*, which is also called a *thunk*. A suspension of type `'a susp` represents a value of type `'a` that is produced by an action. The action is not performed until the suspension's value is *demanded* by function `force`. The action itself is represented by a function of type `unit -> 'a`. The suspension is created by passing the action to the function `delay`; at that point, the action is "pending." If `force` is never called, the action is never performed and remains pending. The first time `force` is called, the action is performed, and the suspension saves the result that is produced. If `force` is called multiple times, the action is still performed just once—later calls to `force` don't repeat the action but simply return the value previously produced.

To implement suspensions, I use a standard combination of imperative and functional code. A suspension is a reference to an `action`, which can be pending or can have produced a result. Functions `delay` and `force` convert to and from suspensions.

646 ⟨*suspensions* 646⟩≡ (644)

```
datatype 'a action
  = PENDING  of unit -> 'a
  | PRODUCED of 'a

type 'a susp = 'a action ref

fun delay f = ref (PENDING f)
fun force cell =
  case !cell
    of PENDING f =>  let val result = f ()
                     in  (cell := PRODUCED result; result)
                     end
     | PRODUCED v => v
```

```
delay : (unit -> 'a) -> 'a susp
force : 'a susp -> 'a
```

## D.1.2 Streams: results of a sequence of actions

An interpreter has to perform not just one action but a whole sequence. If the goal is to read definitions, then the low-level action on top of which other actions are built is "read a line of input." But an interactive interpreter doesn't just read all the input and then convert it all to definitions. Instead, it reads just as much input as is needed to make the first definition, then evaluates the definition and prints the result. To orchestrate all these actions, I use *streams*.

A stream behaves much like a list, except that the first time we look at each element, some action might be taken. And unlike a list, a stream can be infinite. My code uses streams of lines, streams of characters, streams of definitions, and even streams of source-code locations. In this section I define streams and a large collection of related utility functions. Many of the utility functions are directly inspired by list functions like `map`, `filter`, `concat`, `zip`, and `foldl`.

### Stream representation and basic functions

My representation of streams uses three cases:[2]

- The `EOS` constructor represents an empty stream.
- The `:::` constructor (pronounced "cons"), which I intend should remind you of ML's `::` constructor for lists, represents a stream in which an action has already been taken, and the first element of the stream is available (as are the remaining elements). Like the standard `::` constructor, the `:::` is infix.
- The `SUSPENDED` constructor represents a stream in which the action need to produce the next element may not have been taken yet. Getting the element will require forcing the suspension, and if the action in the suspension is pending, it will be taken at that time.

647a ⟨*streams* 647a⟩≡ (644) 647b▷

```
datatype 'a stream
  = EOS
  | :::       of 'a * 'a stream
  | SUSPENDED of 'a stream susp
infixr 3 :::
```

Even though its representation uses mutable state (the suspension), the stream is an immutable abstraction.[3] To observe that abstraction, call `streamGet`. This function performs whatever actions are needed either to produce a pair holding an element an a stream (represented as `SOME (`$x$`, `$xs$`)` or to decide that the stream is empty and no more elements can be produced (represented as `NONE`).

647b ⟨*streams* 647a⟩+≡ (644) ◁647a 647c▷

```
streamGet : 'a stream -> ('a * 'a stream) option
```

```
fun streamGet EOS = NONE
  | streamGet (x ::: xs)     = SOME (x, xs)
  | streamGet (SUSPENDED s) = streamGet (force s)
```

The simplest way to create a stream is by using the `:::` or `EOS` constructors. It can also be convenient to create a stream from a list. When such a streams is read, no new actions are performed.

647c ⟨*streams* 647a⟩+≡ (644) ◁647b 647d▷

```
streamOfList : 'a list -> 'a stream
```

```
fun streamOfList xs =
  foldr (op :::) EOS xs
```

Function `listOfStream` creates a list from a stream. It is useful for debugging.

647d ⟨*streams* 647a⟩+≡ (644) ◁647c 647e▷

```
listOfStream : 'a stream -> 'a list
```

```
fun listOfStream xs =
  case streamGet xs
    of NONE => []
     | SOME (x, xs) => x :: listOfStream xs
```



The more interesting streams are those that result from actions. To help create such stream, I define `delayedStream` as a convenience abbreviation for creating a stream from one action.

647e ⟨*streams* 647a⟩+≡ (644) ◁647d 648a▷

```
delayedStream : (unit -> 'a stream) -> 'a stream
```

```
fun delayedStream action =
  SUSPENDED (delay action)
```

[2] There are simpler representations; this one has the merit that one can define a polymorphic empty stream without running afoul of the value restriction.

[3] To help with debugging, I sometimes violate the abstraction and look at the state of a `SUSPENDED` stream.

### Creating streams using actions and functions

Function `streamOfEffects` produces the stream of results obtained by repeatedly performing a single action (like reading a line of input). The action has type `unit -> 'a option`, and the stream performs the action repeatedly until it returns `NONE`.

648a ⟨*streams* 647a⟩+≡ (644) ◁647e 648b▷

```
fun streamOfEffects next =
  delayedStream (fn () => case next () of NONE => EOS
                                        | SOME a => a ::: streamOfEffects next)
```

`streamOfEffects : (unit -> 'a option) -> 'a stream`

I use `streamOfEffects` to produce a stream of lines from an input file:

648b ⟨*streams* 647a⟩+≡ (644) ◁648a 648c▷

```
type line = string
fun streamOfLines infile =
  streamOfEffects (fn () => TextIO.inputLine infile)
```

`type line`
`streamOfLines : TextIO.instream -> line stream`

Where `streamOfEffects` produces the results of repeating a single *action* again and again, `streamRepeat` simply repeats a single *value* again and again. This operation might sound useless, but here's an example: suppose we read a sequence of lines from a file, and for error reporting, we want to tag each line with its source location, i.e., file name and line number. Well, the file names are all the same, and one easy way to associate the same file name with every line is to repeat the file name indefinitely, then join the two streams. Function `streamRepeat` creates an infinite stream that repeats a value of any type:

648c ⟨*streams* 647a⟩+≡ (644) ◁648b 648d▷

```
fun streamRepeat x =
  delayedStream (fn () => x ::: streamRepeat x)
```

`streamRepeat : 'a -> 'a stream`

A more sophisticated way to produce a stream is to use a function that depends on an evolving *state* of some unknown type `'b`. The function is applied to a state (of type `'b`) and may produce a pair containing a value of type `'a` and a new state. By repeatedly applying the function, we produce a sequence of results of type `'a`. This operation, in which a function is used to expand a value into a sequence, is the dual of the *fold* operation, which is used to collapse a sequence into a value. The new operation is therefore called *unfold*.

648d ⟨*streams* 647a⟩+≡ (644) ◁648c 649a▷

`streamOfUnfold : ('b -> ('a * 'b) option) -> 'b -> 'a stream`

```
fun streamOfUnfold next state =
  delayedStream (fn () => case next state
                            of NONE => EOS
                             | SOME (a, state) => a ::: streamOfUnfold next state)
```

`:::` 647a
`delayedStream` 647e
`EOS` 647a

Function `streamOfUnfold` can turn any "get" function into a stream. In fact, the standard unfold and get operations should obey the following algebraic law:

$$\texttt{streamOfUnfold streamGet } xs \equiv xs.$$

Another useful "get" function is `(fn n => SOME (n, n+1))`; passing this function to `streamOfUnfold` results in an infinite stream of increasing integers.

(Streams, like lists, support not only unfolding but also folding. Function `streamFold` is defined below in chunk 650a.)

### Attaching extra actions to streams

A stream built with `streamOfEffects` or `streamOfLines` has an imperative action built in. But in an interactive interpreter, the action of reading a line should be preceded by another action: printing the prompt. And deciding just what prompt to print requires orchestrating other actions. One option, which I use below, is to attach an imperative action to a "get" function used with `streamOfUnfold`. Another option, which is sometimes easier to understand, is to attach an action to the stream itself. Such an action could reasonable be performed either before or after the action of getting an element from the stream.

Given an action called `pre` and a stream *xs*, I define a stream `preStream (pre, xs)` that adds `pre ()` to the action performed by the stream. Roughly speaking,

`streamGet (preStream (pre, xs))` = `(pre (); streamGet xs)`.

(The equivalence is only rough because the `pre` action is performed only when an action is needed to get a value from *xs*.)

649a ⟨*streams* 647a⟩+≡ (644) ◁648d 649b▷

```
fun preStream (pre, xs) =
  streamOfUnfold (fn xs => (pre (); streamGet xs)) xs
```

`preStream : (unit -> unit) * 'a stream -> 'a stream`

It's also useful to be able to perform an action immediately *after* getting an element from a stream. In `postStream`, I perform the action only if the `streamGet` operation is successful. That way, the `post` action has access to the element that has been gotten. Post-get actions are especially useful for debugging.

649b ⟨*streams* 647a⟩+≡ (644) ◁649a 649c▷

```
fun postStream (xs, post) =
  streamOfUnfold (fn xs => case streamGet xs
                             of NONE => NONE
                              | head as SOME (x, _) => (post x; head)) xs
```

`postStream : 'a stream * ('a -> unit) -> 'a stream`

### Standard list functions ported to streams

Functions like `map`, `filter`, `fold`, `zip`, and `concat` are every bit as useful on streams as they are on lists. streams.

649c ⟨*streams* 647a⟩+≡ (644) ◁649b 649d▷

```
fun streamMap f xs =
  delayedStream (fn () => case streamGet xs
                            of NONE => EOS
                             | SOME (x, xs) => f x ::: streamMap f xs)
```

`streamMap : ('a -> 'b) -> 'a stream -> 'b stream`

649d ⟨*streams* 647a⟩+≡ (644) ◁649c 650a▷

```
fun streamFilter p xs =
  delayedStream (fn () => case streamGet xs
                            of NONE => EOS
                             | SOME (x, xs) => if p x then x ::: streamFilter p xs
                                               else streamFilter p xs)
```

`streamFilter : ('a -> bool) -> 'a stream -> 'a stream`


::: 647a
delayedStream 647e
EOS 647a
streamGet 647b
streamOfUnfold 648d

The only sensible order in which to fold the elements of a stream is the order in which the actions are taken and the results are produced: from left to right:

650a ⟨*streams* 647a⟩+≡ (644) ◁649d 650b▷

```
streamFold : ('a * 'b -> 'b) -> 'b -> 'a stream -> 'b
fun streamFold f z xs =
  case streamGet xs of NONE => z
                     | SOME (x, xs) => streamFold f (f (x, z)) xs
```

Function `streamZip` returns a stream that is as long as the shorter of the two argument streams. In particular, if `streamZip` is applied to a finite stream and an infinite stream, the result is a finite stream.

650b ⟨*streams* 647a⟩+≡ (644) ◁650a 650c▷

```
streamZip : 'a stream * 'b stream -> ('a * 'b) stream
fun streamZip (xs, ys) =
  delayedStream
  (fn () => case (streamGet xs, streamGet ys)
              of (SOME (x, xs), SOME (y, ys)) => (x, y) ::: streamZip (xs, ys)
               | _ => EOS)
```

Concatenation turns a stream of streams of $A$'s into a single stream of $A$'s. I define it using a `streamOfUnfold` with a two-part state: the first element of the state holds an initial `xs`, and the second part holds the stream of all remaining streams, `xss`. To concatenate the stream of streams `xss`, I use an initial state of `(EOS, xss)`.

650c ⟨*streams* 647a⟩+≡ (644) ◁650b 650d▷

```
streamConcat : 'a stream stream -> 'a stream
fun streamConcat xss =
  let fun get (xs, xss) =
        case streamGet xs
          of SOME (x, xs) => SOME (x, (xs, xss))
           | NONE => case streamGet xss
                       of SOME (xs, xss) => get (xs, xss)
                        | NONE => NONE
  in  streamOfUnfold get (EOS, xss)
  end
```

The composition of `concat` with `map f` is very common in list and stream processing, so I give it a name.

```
:::          647a
delayedStream
             647e
EOS          647a
streamGet    647b
streamMap    649c
streamOfList
             647c
streamOfUnfold
             648d
```

⟨*streams* 647a⟩+≡ (644) ◁650c 650e▷

```
streamConcatMap : ('a -> 'b stream) -> 'a stream -> 'b stream
fun streamConcatMap f xs = streamConcat (streamMap f xs)
```

The code used to append two streams is much like the code used to concatenate arbitrarily many streams. To avoid duplicating the tricky manipulation of states, I simply implement append using concatenation.

⟨*streams* 647a⟩+≡ (644) ◁650d

```
@@@ : 'a stream * 'a stream -> 'a stream
infix 5 @@@
fun xs @@@ xs' = streamConcat (streamOfList [xs, xs'])
```

## D.2 Managing errors detected during parsing

Our goal is to build parsers that proceed by examining sequences of alternatives. When I ask, "is the next line of input a `val`?", I expect one of three answers:

- Yes, it's a `val`. Our work here is done.
- No, it's not a `val`. Try something else.
- It looks like it tried to be a `val`, but something went terribly wrong. For example,

```
-> (val x)
error: expected (val x e) in standard input, line 1
-> (val x 1 2)
error: expected (val x e) in standard input, line 2
```

In that third situation, when an error is detected, it's up to the parser to identify the error and to recover as quickly and cleanly as it can. To do so, it must be aware of errors. That means that my parsing combinators—the functions I use to build parsers—must also be aware of errors. There is a large body of literature on using functional-programming techniques to manage errors. In this section I present some code based on those techniques.[4]

A function that's guaranteed always to produce a result of type `'a` simply returns such a result. A function that might produce a result of type `'a` or might detect an error returns a result of type `'a error`. The result contains either a value of type `'a` or an error message.

651a ⟨*error handling* 651a⟩≡ (644) 651b▷

```
datatype 'a error = OK of 'a | ERROR of string
```

How do we compose error-detecting functions? That is, how to we write `g (f x)` in the case where either `f` or `g` might detect an error? The standard technique is to define a sequencing operator written `>>=`, which uses a special form of continuation-passing style. (The traditional name of the `>>=` operator is "bind," but you might wish to pronounce it "and then.") The idea is that we apply `f` to `x`, and if the result is `OK y`, we can then continue by applying `g` to `y`. But if the result of applying `(f x)` is an error, that error is the result of the whole computation. The `>>=` operator sequences the possibly erroneous result `(f x)` with the continuation `g`, thus

```
f x >>= g
```

In the definition, I write the second function as `k`, not `g`, because `k` is the traditional letter for a continuation.

651b ⟨*error handling* 651a⟩+≡ (644) ◁651a 652a▷

```
infix 1 >>=
fun (OK x)      >>= k  =  k x
  | (ERROR msg) >>= k  =  ERROR msg
```

```
>>= : 'a error * ('a -> 'b error) -> 'b error
```

[4] Many ML programs do not use the techniques I define in this section; instead they single errors by raising exceptions. The problem with exceptions is that it's not always easy to predict exactly when they will be raised, and it's not always easy to know where to put the handlers. By making an error result a special type of value, we enable the type system to help us make sure errors are handled.

A very common special case occurs when the continuation always succeeds. That is, the idea is that if `(f x)` succeeds, apply `k'` to it; otherwise propagate the error. I know of no standard way to write this operator,[5], so I've chosen `>>=+`, which you might also choose to pronounce "and then."

652a ⟨*error handling* 651a⟩+≡ (644) ◁651b 652b▷

```
infix 1 >>=+
fun e >>=+ k'  =  e >>= OK o k'
```

`>>=+ : 'a error * ('a -> 'b) -> 'b error`

Sometimes a whole list of results are checked for errors independently and then must be combined. I call the combining operation `errorList`.[6] I implement it by folding over the list of possibly erroneous results, combining *all* error messages.

652b ⟨*error handling* 651a⟩+≡ (644) ◁652a

```
fun errorList es =
  let fun cons (OK x, OK xs) = OK (x :: xs)
        | cons (ERROR m1, ERROR m2) = ERROR (m1 ^ "; " ^ m2)
        | cons (ERROR m, OK _) = ERROR m
        | cons (OK _, ERROR m) = ERROR m
  in  foldr cons (OK []) es
  end
```

`errorList : 'a error list -> 'a list error`

## D.3 Stream transformers, which act as parsers

Our ultimate goal is to turn streams of input lines into streams of definitions. Along the way we may also have streams of characters, "tokens," types, expressions, and more. To handle all these different kinds of streams using a single set of operators, I define a type representing a *stream transformer*. A stream transformer from $A$ to $B$ takes a stream of $A$'s as input and either succeeds, fails, or detects an error:

- If it succeeds, it consumes *zero or more* $A$'s from the input stream and produces exactly one $B$. It returns a pair containing `OK` $B$ plus whatever $A$'s were not consumed.
- If it fails, it returns `NONE`.
- If it detects an error, it returns a pair containing `ERROR` $m$, where $m$ is a message, plus whatever $A$'s were not consumed.

652c ⟨*parsing combinators* 652c⟩≡ (644) 653a▷

```
type ('a, 'b) xformer =
  'a stream -> ('b error * 'a stream) option
```

`type ('a, 'b) xformer`

If we apply `streamOfUnfold` to an `('a, 'b) xformer`, we get a function that maps a stream of $A$'s to a stream of $B$'s-with-error.

The stream-transformer abstraction supports many, many operations. These operations, known as *parsing combinators*, have been refined by functional programmers for over two decades, and they can be expressed in a variety of guises. The guise I have chosen uses notation from *applicative functors* and from the ParSec parsing library.


>>= 651b
ERROR 651a
type error 651a
OK 651a
type stream 647a


[5] Haskell uses `flip fmap`.
[6] Haskell calls it `sequence`.

I begin very abstractly, by presenting combinators that don't actually consume any inputs. The next two sections present only "constant" transformers and "glue" functions that build transformers from other transformers. With those functions in place, we proceed to real, working parsing combinators. These combinators are split into two groups: "universal" combinators that work with any stream, and "parsing" combinators that expect a stream of tokens with source-code locations.

### D.3.1 Error-free transformers and their composition

The `pure` combinator takes a $B$ as argument and returns an $A$-to-$B$ transformer that consumes no $A$'s as input and produces the given $B$:

653a ⟨*parsing combinators* 652c⟩+≡ (644) ◁652c 653b▷

```
fun pure y = fn xs => SOME (OK y, xs)
```

```
pure : 'b -> ('a, 'b) xformer
```

The ultimate role of parser is to read inputs in sequence. Such a parser is a stream transformer, and it is made by composing smaller stream transformers that read parts of the input. The sequential composition operator, if you have not seen it before, may look quite strange. To compose `tx_f` and `tx_b` in sequence, you use the infix operator `<*>`, which is pronounced "applied to." The composition is written `tx_f <*> tx_b`, and here's how it works:

1. First `tx_f` reads some $A$'s and produces a *function* `f` of type $B \to C$.
2. Next `tx_b` reads some more $A$'s and produces a value `y` which is a $B$.
3. The combination `tx_f <*> tx_b` reads no more input but simply applies `f` to `y` and returns that value (a $C$) as its result.

This idea may seem borderline insane. How can reading a sequence of $A$'s produce a function? The secret is that almost always, the function is produced by `pure`, without actually reading any $A$'s. But it's a great way to do business, because when the parser is written using the `pure` and `<*>` combinators, the code resembles a Curried function application.

For the combination `tx_f <*> tx_b` to succeed, both `tx_f` and `tx_b` must succeed. So I use nested case analysis.

653b ⟨*parsing combinators* 652c⟩+≡ (644) ◁653a 653c▷

```
infix 3 <*>
```

```
<*> : ('a, 'b -> 'c) xformer * ('a, 'b) xformer -> ('a, 'c) xformer
```

```
fun tx_f <*> tx_b =
  fn xs => case tx_f xs
             of NONE => NONE
              | SOME (ERROR msg, xs) => SOME (ERROR msg, xs)
              | SOME (OK f, xs) =>
                  case tx_b xs
                    of NONE => NONE
                     | SOME (ERROR msg, xs) => SOME (ERROR msg, xs)
                     | SOME (OK y, xs) => SOME (OK (f y), xs)
```

ERROR 651a
OK 651a

The common case of creating `tx_f` using `pure` has a special operator `<$>`, which is also pronounced "applied to." It combines a $B$-to-$C$ function with an $A$-to-$B$ transformer to produce an $A$-to-$C$ transformer.

653c ⟨*parsing combinators* 652c⟩+≡ (644) ◁653b 654a▷

```
infixr 4 <$>
```

```
<$> : ('b -> 'c) * ('a, 'b) xformer -> ('a, 'c) xformer
```

```
fun f <$> p = pure f <*> p
```

There are a variety of ways to create useful functions in the `f` position. Many such functions are Curried. Here are some of them.

654a ⟨*parsing combinators* 652c⟩+≡ (644) ◁653c 654b▷

```
fst    : ('a * 'b) -> 'a
snd    : ('a * 'b) -> 'b
pair   : 'a -> 'b -> 'a * 'b
curry  : ('a * 'b -> 'c) -> ('a -> 'b -> 'c)
curry3 : ('a * 'b * 'c -> 'd) -> ('a -> 'b -> 'c -> 'd)
```

```
fun id x = x
fun fst (x, y) = x
fun snd (x, y) = y
fun pair x y = (x, y)
fun curry  f x y   = f (x, y)
fun curry3 f x y z = f (x, y, z)
```

As an example, if `name` parses a name and `exp` parses an expression then in a `let` binding we can parse a `name * exp` pair by

```
pair <$> name <*> exp
```

(To parse $\mu$Scheme, we would need also to parse the surrounding parentheses.) As another example, if in $\mu$Scheme we have seen the keyword `if`, we can follow it by the parser

```
curry3 IFX <$> exp <*> exp <*> exp
```

which creates the syntax for an `if` expression.

The combinator `<*>` creates parsers that read things in sequence; but it can't make a choice. If any parser in the sequence fails, the whole sequence fails. To make a choice, as in "`val` or expression or `define` or `use`," we use a choice operator. The choice operator is written `<|>` and pronounced "or." If `t1` and `t2` are both $A$-to-$B$ transformers, then `t1 <|> t2` is an $A$-to-$B$ transformer that first tries `t1`, then tries `t2`, succeeding if either succeeds, detecting an error if either detects an error, and failing only if both fail. To assure that the result has a predictable type no matter which transformer is used, both `t1` and `t2` have to have the same type.

654b ⟨*parsing combinators* 652c⟩+≡ (644) ◁654a 655a▷

```
<|> : ('a, 'b) xformer * ('a, 'b) xformer -> ('a, 'b) xformer
```

```
infix 1 <|>
fun t1 <|> t2 = (fn xs => case t1 xs of SOME y => SOME y | NONE => t2 xs)
```

## D.3.2 Ignoring results produced by transformers

If a parser sees the stream of tokens

`(` `if` `(` `<` `x` `y` `)` `x` `y` `)`,

we want it to build an abstract-syntax tree using `IFX` and three expressions. The parentheses and keyword `if` serve to identify the `if`-expression and to make sure it is well formed, so we do need to read them from the input, but we don't need to do anything with the results that are produced. Using a parser and then ignoring the result is such a common operation that special abbreviations have evolved to support it.

The abbreviations are formed by modifying the `<*>` or `<$>` operator to remove the angle bracket on the side containing the result we don't care about. For example,

- Parser `p1 <* p2` reads the input of `p1` and then the input of `p2`, but it returns only the result of `p1`.
- Parser `p1 *> p2` reads the input of `p1` and then the input of `p2`, but it returns only the result of `p2`.
- Parser `v <$ p` parses the input the way `p` does, but it then ignores `p`'s result and instead produces the value `v`.

655a ⟨*parsing combinators* 652c⟩+≡ (644) ◁654b 655b▷

```
<*  : ('a, 'b) xformer * ('a, 'c) xformer -> ('a, 'b) xformer
 *> : ('a, 'b) xformer * ('a, 'c) xformer -> ('a, 'c) xformer
<$  : 'b               * ('a, 'c) xformer -> ('a, 'b) xformer
```

```
infix 3 <* *>
fun p1 <*  p2 = curry fst <$> p1 <*> p2
fun p1  *> p2 = curry snd <$> p1 <*> p2

infixr 4 <$
fun v <$ p = (fn _ => v) <$> p
```

## D.3.3 Finally, transformers that look at the input stream

None of the transformers above looks directly at an input stream. The fundamental operations are `pure`, `<*>`, and `<|>`; `pure` never looks at the input, and `<*>` and `<|>` simply sequence or alternate between other parsers which do the actual looking. It's time to meet those parsers.

The simplest input-inspecting parser is `one`. It's an $A$-to-$A$ transformer that succeeds if and only if there is a value in the input. If there's no value input, `one` fails; it never signals an error.

655b ⟨*parsing combinators* 652c⟩+≡ (644) ◁655a 655c▷

```
one : ('a, 'a) xformer
```

```
fun one xs = case streamGet xs
               of NONE => NONE
                | SOME (x, xs) => SOME (OK x, xs)
```

The counterpart of `one` is a parser that succeeds if and only if there is *no* input—that is, if we have reached the end of the stream. This parser, which is called `eos`, can produce no useful result, so it produces the empty tuple, which has type `unit`.

655c ⟨*parsing combinators* 652c⟩+≡ (644) ◁655b 655d▷

```
eos : ('a, unit) xformer
```

```
fun eos xs = case streamGet xs
               of NONE => SOME (OK (), EOS)
                | SOME _ => NONE
```


<$> 653c
<*> 653b
curry 654a
EOS 647a
fst 654a
OK 651a
snd 654a
streamGet 647b


SHOCKINGLY, THAT'S IT. WE HOPE THESE ARE THE ONLY THINGS THAT CALL STREAM-GET.

It can also be useful to peek at the contents of a stream, without looking at any input, and while ignoring errors.

655d ⟨*parsing combinators* 652c⟩+≡ (644) ◁655c 656a▷

```
peek : 'a xformer -> 'a stream -> 'b option
```

```
fun peek tx xs = case tx xs of SOME (OK (a, _), _) => SOME a
                             | _ => NONE
```

And we might want to transform some input, then rewind it back to the starting point. (Actions can't be undone, but at least the input can be read again.)

656a ⟨*parsing combinators* 652c⟩+≡ (644) ◁655d 656b▷

```
fun rewind tx xs = case tx xs of SOME (ay, _) => SOME (ay, xs)
                               | NONE => NONE
```

rewind : ('a, 'b) xformer -> ('a, 'b) xformer

## D.3.4 Parsing combinators

Real parsers largely build on `<$>`, `<*>`, `<|>`, and `one` by adding the following ideas:

- Perhaps we'd like to succeed only if an input satisfies certain conditions. For example, if we're trying to read a number, we might want to write a character parser that succeeds only when the character is a digit.
- Most utterances in programming languages are made by composing things in sequence. For example, in $\mu$Scheme, the characters in an identifier are a nonempty sequence of "ordinary" characters. And the arguments in a function application are a possibly empty sequence of expressions.
- Although I've avoided using "optional" syntax in my own designs, many, many programming languages do use constructs in which parts are optional. For example, in C, the use of an `else` clause with an `if` statement is optional.

This section presents standard parsing combinators that help implement conditional parsers, parsers for sequences, and parsers for optional syntax.

### Parsers based on conditions

Combinator `sat` wraps an $A$-to-$B$ transformer with a $B$-predicate such that the wrapped transformer succeeds only when the underlying transformer succeeds and produces a value that satisfies the predicate.

656b ⟨*parsing combinators* 652c⟩+≡ (644) ◁656a 656c▷

```
fun sat p tx xs =
  case tx xs
    of answer as SOME (OK y, xs) => if p y then answer else NONE
     | answer => answer
```

sat : ('b -> bool) -> ('a, 'b) xformer -> ('a, 'b) xformer

Transformer `oneEq x` is an $A$-to-$A$ transformer that succeeds if and only if there is a next input and that input equals `x`. It is typically used to recognize special characters like parentheses and minus signs.

OK 651a
one 655b

⟨*parsing combinators* 652c⟩+≡ (644) ◁656b 657a▷

```
fun oneEq x = sat (fn x' => x = x') one
```

oneEq : ''a -> (''a, ''a) xformer

A more subtle condition is that a partial function can turn an input into something we're looking for. If we have an $A$-to-$B$ transformer, and we compose it with a function that given a $B$, sometimes produces a $C$, then we get an $A$-to-$C$ transformer. Because there's a close analogy with the application operator `<$>`, I'm notating this *partial* application operator as `<$>?`, with a question mark.

657a ⟨*parsing combinators* 652c⟩+≡ (644) ◁656c 657b▷

```
infixr 4 <$>?
fun f <$>? tx =
  fn xs => case tx xs
             of NONE => NONE
              | SOME (ERROR msg, xs) => SOME (ERROR msg, xs)
              | SOME (OK y, xs) =>
                  case f y
                    of NONE => NONE
                     | SOME z => SOME (OK z, xs)
```

```
<$>? : ('b -> 'c option) * ('a, 'b) xformer -> ('a, 'c) xformer
```

We can run a parser conditional on the success of another parser. Parser `t1 <&> t2` succeeds only if both `t1` and `t2` succeed at the same point. This parser looks at enough input to decide if `t1` succeeds, but it does not consume that input—it consumes only the input of `t2`. It's usually used to detect errors, as in "when you see something bad, signal an error."

657b ⟨*parsing combinators* 652c⟩+≡ (644) ◁657a 657c▷

```
infix 3 <&>
fun t1 <&> t2 = fn xs =>
  case t1 xs
    of SOME (OK _, _) => t2 xs
     | SOME (ERROR _, _) => NONE
     | NONE => NONE
```

```
<&> : ('a, 'b) xformer * ('a, 'c) xformer -> ('a, 'c) xformer
```

We can also use the success or failure of a parser as a condition. Parser `notFollowedBy t` succeeds if and only if `t` fails. Parser `notFollowedBy t` may *look* at the input, but it never *consumes* any input. I use `notFollowedBy` when reading integer literals, to make sure that the digits are not followed by a letter or other non-delimiting symbol.

657c ⟨*parsing combinators* 652c⟩+≡ (644) ◁657b 657d▷

```
fun notFollowedBy t xs =
  case t xs
    of NONE => SOME (OK (), xs)
     | SOME _ => NONE
```

```
notFollowedBy : ('a, 'b) xformer -> ('a, unit) xformer
```

We now have something that resembles a little Boolean algebra for parsers: functions `when`, `<|>`, and `notFollowedBy` play the roles of "and," "or," and "not."



### Parsers for sequences

Inputs are full of sequences. A function takes a sequence of arguments, a program is a sequence of definitions, and a method definition contains a sequence of expressions. To create transformers that process sequences, we define function `many` and `many1`. If `t` is an $A$-to-$B$ transformer, then `many t` is an $A$-to-list-of-$B$ transformer. It runs `t` as many times as possible. And even if `t` fails, `many t` always succeeds: when `t` fails, `many t` returns an empty list of $B$'s.

657d ⟨*parsing combinators* 652c⟩+≡ (644) ◁657c 658a▷

```
fun many t =
  curry (op ::) <$> t <*> (fn xs => many t xs) <|> pure []
```

```
many : ('a, 'b) xformer -> ('a, 'b list) xformer
```

I'd really like to write that first alternative as

```
curry (op ::) <$> t <*> many t
```

but that formulation leads to instant death by infinite recursion. If you write your own parsers, it's a problem to watch out for.

Sometimes an empty list isn't acceptable. In that case, use `many1 t`, which succeeds only if `t` succeeds at least once.

658a ⟨*parsing combinators* 652c⟩+≡ (644) ◁657d 658b▷

```
many1 : ('a, 'b) xformer -> ('a, 'b list) xformer
fun many1 t =
  curry (op ::) <$> t <*> many t
```

Although `many t` always succeeds, `many1 t` can fail.

Both `many` and `many1` are "greedy;" that is, they repeat `t` as many times as possible. Client code has to be careful to ensure that calls to `many` and `many1` terminate. As it stands, if `t` can succeed without consuming any input, then `many t` does not terminate, so it is an unchecked run-time error to pass `many` a transformer that succeeds without consuming input. The same goes for `many1`.

Sometimes instead of zero, one, or many $B$'s, we just one zero or one; such a $B$ might be called "optional." For example, a numeric literal begins with an optional minus sign. Function `optional` turns an $A$-to-$B$ transformer into an $A$-to-optional-$B$ transformer. Like `many t`, `optional t` always succeeds.

658b ⟨*parsing combinators* 652c⟩+≡ (644) ◁658a 658c▷

```
optional : ('a, 'b) xformer -> ('a, 'b option) xformer
fun optional t =
  SOME <$> t <|> pure NONE
```

Transformers made with `many` and `optional` succeed even when there is no input. They also succeed when there is input that they don't recognize.

### D.3.5 Error-detecting transformers and their composition

Sometimes an error is detected not by a parser but by a function that is applied to the results of parsing. A classic example is a function definition: if the formal parameters are syntactically correct but contain duplicate name, an error should be signalled. We would transform the input into a value of type `name list error`. But the transformer type already includes the possibility of error, and we would prefer that errors detected by functions be on the same footing as errors detected by parsers, and that they be handled by the same mechanisms. To enable such handling, I define `<*>!` and `<$>!` combinator that merge function-detected errors with parser-detected errors.


<$> 653c
<*> 653b
<|> 654b
curry 654a
ERROR 651a
many 657d
OK 651a
pure 653a


⟨*parsing combinators* 652c⟩+≡ (644) ◁658b

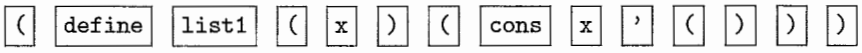
```
<*>! : ('a, 'b -> 'c error) xformer * ('a, 'b) xformer -> ('a, 'c) xformer
<$>! : ('b -> 'c error)             * ('a, 'b) xformer -> ('a, 'c) xformer

infix 2 <*>!
fun tx_ef <*>! tx_x =
  fn xs => case (tx_ef <*> tx_x) xs
             of NONE => NONE
              | SOME (OK (OK y),      xs) => SOME (OK y,      xs)
              | SOME (OK (ERROR msg), xs) => SOME (ERROR msg, xs)
              | SOME (ERROR msg,      xs) => SOME (ERROR msg, xs)
infixr 4 <$>!
fun ef <$>! tx_x = pure ef <*>! tx_x
```

## D.4 Lexical analyzers: transformers of characters

The interpreters in this book consume one line at a time. But characters *within* a line may be split into multiple *tokens*. For example, the line

```
(define list1 (x) (cons x '()))
```

should be split into the tokens




This section reusable transformers that are specialized to transform streams of characters into something else, usually tokens.

659a ⟨*support for lexical analysis* 659a⟩≡ (644) 659b▷

```
type 'a lexer = (char, 'a) xformer
```

type 'a lexer

The type `'a lexer` should be pronounced "lexer returning `'a`."

In popular languages, a character like a semicolon or comma usually does not join with other tokens to form a character. In this book, the left and right parentheses keep to themselves and don't group with other characters. And in just about every non-esoteric language, blank space separates tokens. A character whose presence marks the end of one token (and possibly the beginning of the next) is called a *delimiter*. In this book, whitespace and parentheses are the main delimiter characters. The semicolon, which introduces a comment, also acts as a delimiter.

659b ⟨*support for lexical analysis* 659a⟩+≡ (644) ◁659a 659c▷

```
fun isDelim c = Char.isSpace c orelse Char.contains "();" c
```

isDelim : char -> bool

`Char.isSpace` recognizes all whitespace characters. `Char.contains` takes a string and a character and says if the string contains the character. These functions are in the initial basis of Standard ML.

All languages in this book ignore whitespace. Lexer `whitespace` is typically combined with another lexer using the `*>` operator.

659c ⟨*support for lexical analysis* 659a⟩+≡ (644) ◁659b 660a▷

```
val whitespace = many (sat Char.isSpace one)
```

whitespace : char list lexer

Most languages in this book are, like Scheme, very liberal about names. Just about any sequence of characters, as long as it is free of delimiters, can form a name. But there's one big exception: a sequence of digits doesn't form a name; it forms an integer literal. Because integer literals offer several complications, and because they are used in all the languages in this book, it makes sense to deal with the complications in one place: here.

many 657d
one 655b
sat 656b

The rules for integer literals are as follows:

- The integer literal may begin with a minus sign.
- It continues with one or more digits.
- If it is followed by character, that character must be a delimiter. (In other words, it must not be followed by a non-delimiter.)
- When the sequence of digits is converted to an `int`, the arithmetic used in the conversion must not overflow.

Function `intChars` does the lexical analysis to grab the characters; `intFromChars` handles the conversion and its potential overflow, and `intToken` puts everything together. Because not every language has the same delimiters, both `intChars` and `intToken` receive a predicate that identifies delimiters.

660a ⟨*support for lexical analysis* 659a⟩+≡ (644) ◁659c 660b▷

```
fun intChars isDelim =
  (curry (op ::) <$> oneEq #"-" <|> pure id) <*> many1 (sat Char.isDigit one) <*
  notFollowedBy (sat (not o isDelim) one)
```

```
intChars : (char -> bool) -> char list lexer
```

Function `Char.isDigit`, like `Char.isSpace`, is part of Standard ML.

Function `intFromChars` works by combining three functions from Standard ML's initial basis. Function `implode` converts to string; `Int.fromString` converts to `int option` (raising `Overflow` if the literal is too big); and `valOf` converts from `int option` to `int`. The `~` function, which is used when we see a minus sign, is ML's way of writing negation.

660b ⟨*support for lexical analysis* 659a⟩+≡ (644) ◁660a 660c▷

```
fun intFromChars (#"-" :: cs) =
      intFromChars cs >>=+ ~
  | intFromChars cs =
      (OK o valOf o Int.fromString o implode) cs
      handle Overflow => ERROR "this interpreter can't read arbitrarily large integers"
```

```
intFromChars : char list -> int error
```

In this book, every language except $\mu$Prolog can use `intToken`.

⟨*support for lexical analysis* 659a⟩+≡ (644) ◁660b

```
fun intToken isDelim =
  intFromChars <$>! intChars isDelim
```

```
intToken : (char -> bool) -> int lexer
```


<$> 653c
<$>! 658c
<*> 653b
<|> 654b
>>=+ 652a
curry 654a
ERROR 651a
id 654a
many1 658a
notFollowedBy 657c
OK 651a
one 655b
oneEq 656c
pure 653a
sat 656b


## D.5 Parsers: reading tokens and source-code locations

To read definitions, expressions, and types, it helps to work at a higher level of abstraction than individual characters. All the parsers in this book use two stages: first a lexer groups characters into tokens, then a parser transforms tokens into syntax. Not all languages use the same tokens, so the code in this section assumes that the type `token` and two related functions are defined. Function `isLiteral` tells us if a token was produced from the given literal string; it is used in parsing. function `tokenString` returns a string representation of any given token; it is used in debugging.

660d ⟨*support for parsing* 660d⟩≡ (644) 661a▷

```
(* token, isLiteral, and
   tokenString can be defined
   differently in each language *)
```

```
type token
isLiteral : string -> token -> bool
tokenString : token -> string
```

YOU CAN SEE EXAMPLE DEFINITIONS WHERE IN WHAT APPENDIX, PLEASE?

In some languages in this book, not all kinds of tokens are used as literals in parsing. For example, integer literals are typically used for their values, not to mark language constructs. For such tokens, it is OK if `isLiteral` always returns `false`.

I hope transforming a stream of characters to a stream of tokens to a stream of definitions sounds appealing—but it simplifies the story a little too much. If nothing ever went wrong, it would be fine if all we ever saw were tokens. But if something does go wrong, I want to be able to do more than throw up my hands:

- I want say *where* things went wrong—at what *source-code location.*
- I want to get rid of the bad tokens that caused the error.
- I want to be able to start parsing over again interactively, without having to kill an interpreter and start over.

To support error reporting and recovery takes a lot of machinery.

## D.5.1 Source-code locations

To be able to say where things go wrong, we need to track *source-code locations.* Compilers that take themselves seriously, which includes all the compilers I have built and most of the ones I have worked on, report source-code locations right down to the individual character: file `broken.c`, line 12, column 17. In production compilers, such precision is admirable. But in a pedagogical interpreter, the desire for precision has to be balanced against the need for simplicity. The best compromise is to track only source file and line number. That's good enough to help programmers find errors, and it eliminates the bookkeeping that would otherwise be needed to track column numbers.

661a ⟨*support for parsing* 660d⟩+≡ (644) ◁660d 661b▷

```
type srcloc = string * int
fun srclocString (source, line) =
  source ^ ", line " ^ Int.toString line
```

type srcloc
srclocString : srcloc -> string

To signal an error at a given location, call `errorAt`.

661b ⟨*support for parsing* 660d⟩+≡ (644) ◁661a 661c▷

```
fun errorAt msg loc =
  ERROR (msg ^ " in " ^ srclocString loc)
```

errorAt : string -> srcloc -> 'a error

We can pair source-code locations with individual lines and tokens. To make it easier to read the types, I define a type abbreviation which says that a value paired with a location is "located."

661c ⟨*support for parsing* 660d⟩+≡ (644) ◁661b 661d▷

```
type 'a located = srcloc * 'a
```

type 'a located

All locations originate in a located stream of lines. The locations share a filename, and the line numbers are $1, 2, 3, \ldots$ and so on.

661d ⟨*support for parsing* 660d⟩+≡ (644) ◁661c 662a▷

```
fun locatedStream (streamname, inputs) =
  let val locations = streamZip (streamRepeat streamname,
                                 streamOfUnfold (fn n => SOME (n, n+1)) 1)
  in  streamZip (locations, inputs)
  end
```

locatedStream : string * line stream -> line located stream

ERROR 651a
streamOfUnfold 648d
streamRepeat 648c
streamZip 650b

### D.5.2 Flushing bad tokens

A standard parser for a batch compiler needs only to see a stream of tokens and to know from what source-code location each token came. A batch compiler can simply read all its input and report all the errors it wants to report.[7] But an interactive interpreter may not use an error as an excuse to read an indefinite amount of input. It must instead bring its error processing to a prompt conclusion and ready itself to read the next line. To do so, it needs to know where the line boundaries are! For example, if I find an error on line 6, I want to read all the tokens on line 6, throw them away, and start over again on line 7. The nasty bit is that I want to do it *without* reading line 7—reading line 7 will take an action and will likely have the side effect of printing a prompt. And I want it to be the correct prompt. I therefore define a new type constructor `inline`. A value of type `'a inline` either contains an value of type `'a` that occurs in a line, or and end-of-line marker. A stream of such values can be drained up to the end of the line.[8]

662a ⟨*support for parsing* 660d⟩+≡ (644) ◁661d

```
type 'a inline
drainLine : 'a inline stream -> 'a inline stream
```

```
datatype 'a inline
  = EOL of int (* number of the line that ends here *)
  | INLINE of 'a

fun drainLine EOS = EOS
  | drainLine (SUSPENDED s)     = drainLine (force s)
  | drainLine (EOL _    ::: xs) = xs
  | drainLine (INLINE _ ::: xs) = drainLine xs
```

With source-code locations and end-of-line markers ready, we are finally able to define parsers.

### D.5.3 Parsing located, in-line tokens

A value of type `'a parser` takes a stream of located tokens set between end-of-line markers, and it returns a value of type `'a`, plus any leftover tokens.

662b ⟨*parsing utilities* 662b⟩≡ (644) 663a▷

```
type 'a parser = (token located inline, 'a) xformer
```

```
type 'a parser
```

```
::: 647a
EOS 647a
force 646
SUSPENDED 647a
```

[7] Batch compilers vary widely in the ambitions of their parsers. Some simple parsers report just one error and stop. Some sophisticated parsers analyze the entire input and report the smallest number of changes needed to make the input syntactically correct. And some ill-mannered parsers become confused after an error and start spraying meaningless error messages. But all of them have access to the entire input. We don't.

[8] At some future point I may need to change `drainLine` to keep the `EOL` in order to track locations in $\mu$Prolog.

The EOL and INLINE constructors are essential for error recovery, but for parsing, they just get in the way. Our first order of business is to define analogs of one and eos that ignore EOL. Parser token takes one token; parser srcloc takes one source-code location; and parser noTokens succeeds only if there are no tokens left in the input. They are built on top of "utility" parsers eol and inline. The two utility parsers have different contracts; eol succeeds only when at EOL, but inline scans past EOL to look for INLINE.

663a ⟨*parsing utilities* 662b⟩+≡ (644) ◁662b 663b▷

```
local
  fun asEol (EOL n) = SOME n
    | asEol (INLINE _) = NONE
  fun asInline (INLINE x) = SOME x
    | asInline (EOL _)    = NONE
in
  fun eol    xs = (asEol <$>? one) xs
  fun inline xs = (many eol *> asInline <$>? one) xs
end

val token    =         snd <$> inline  : token parser
val srcloc   = rewind (fst <$> inline) : srcloc parser
val noTokens = notFollowedBy token : unit parser
```

```
eol      : ('a inline, int) xformer
inline   : ('a inline, 'a)  xformer
token    : token parser
srcloc   : srcloc parser
noTokens : unit parser
```

Sometimes the easiest way to keep track of source-code locations is to pair a source-code location with a result from a parser. This happens just often enough that I find it worth while to define the @@ function. (Associate the word "at" with the idea of "location.") The code uses a dirty trick: it works because srcloc looks at the input but does not consume any tokens.

663b ⟨*parsing utilities* 662b⟩+≡ (644) ◁663a 663c▷

```
fun @@ p = pair <$> srcloc <*> p
```

```
@@ : 'a parser -> 'a located parser
```

## D.5.4 Parsers that report errors

The <?> operator typically follows a list of alternatives, as in the parser for definitions on page . It reports that it couldn't recognize its input, and it gives the source-code location of the unrecognized token. If there is no token, there is no error—at end of file, rather than signal an error, a parser made using <?> fails.

663c ⟨*parsing utilities* 662b⟩+≡ (644) ◁663b 664a▷

```
infix 0 <?>
fun p <?> expected = p <|> errorAt ("expected " ^ expected) <$>! srcloc
```

```
<?> : 'a parser * string -> 'a parser
```

Another common error-detecting technique is to use a parser `p` to detect some input that shouldn't be there. We can't simply combine `p` with `errorAt` and `srcloc` in the same way that `<?>` does, because we have two goals: *consume* the tokens recognized by `p`, and also *report* the error at the location of the first of those tokens. We can't use `errorAt` until *after* `p` succeeds, but we have to use `srcloc` on the input stream as it is *before* `p` is run. My solution is to define a special combinator that keeps a copy of the tokens inspected by `p`. If parser `p` succeeds, then parser `p <!> msg` consumes the tokens consumed by `p` and reports error `msg` at the location of `p`'s first token.

664a ⟨*parsing utilities* 662b⟩+≡ (644) ◁663c 664b▷

```
infix 4 <!>
fun p <!> msg =
  fn tokens => (case p tokens
                  of SOME (OK _, unused) =>
                       (case peek srcloc tokens
                          of SOME loc => SOME (errorAt msg loc, unused)
                           | NONE => NONE)
                   | _ => NONE)
```

```
<!> : 'a parser * string -> 'b parser
```

## D.5.5 Parsers for common programming-language idioms

This section defines special-purpose parsers and combinators which handle phrases and idioms that appear in many of the languages in this book.

### Keywords, brackets, and other literals

It's extremely common to want to recognize literal tokens, like the keyword `if` or a left or right parenthesis. The `literal` parser accepts any token whose concrete syntax is an exact match for the given string argument.

664b ⟨*parsing utilities* 662b⟩+≡ (644) ◁664a 664c▷

```
fun literal s =
  ignore <$> sat (isLiteral s) token
```

```
literal : string -> unit parser
```

Like the type `token`, the function `isLiteral` is different for each programming language in this book.

When it succeeds, the `literal` parser returns the empty tuple, which is not useful for anything. This empty tuple can be ignored by writing parsers like `literal "(" *> p`, but notationally, this is a towering nuisance. Instead, I define new combinators `--<` and `>--` so I can write parsers like `"(" >-- p`. Theses combinators have higher precedence than `<*>`, so they "pull in" literal strings that appear in sequences. As a mnemonic, the angle bracket "swallows" the literal we want to ignore (or if you prefer, it points to what we keep).

⟨*parsing utilities* 662b⟩+≡ (644) ◁664b 665▷

```
infix  6 --<
infixr 7 >--
    (* if we want to mix these operators, they can't have equal precedence *)
fun (a >-- p) = literal a *> p
fun (p --< a) = p <* literal a
```

```
>-- : string    * 'a parser -> 'a parser
--< : 'a parser * string    -> 'a parser
```


<$> 653c
errorAt 661b
isLiteral, in Typed μScheme 671b
in μProlog 710d
in μSmalltalk 699d
OK 651a
peek 655d
sat 656b
srcloc 663a
token 663a

### Bracketed expressions

Almost every language in this book uses parenthesis-prefix syntax (Scheme syntax). The `bracket`[9] function creates a parser that recognizes inputs of the form

( *keyword stuff* )

The `bracket` function embodies some useful error handling:

- It takes an extra parameter `expected`, which says, when anything goes wrong, what the parser was expecting in the way of *stuff*.
- If something does go wrong parsing *stuff*, it calls `scanToCloseParen` to scan past all the tokens where *stuff* was expected, up to and including the matching close parenthesis.

Once the parser sees the opening parenthesis and the keyword, it is committed: either parser `p` parses *stuff* correctly, or there's an error.

665 ⟨*parsing utilities* 662b⟩+≡ (644) ◁664c 666a▷

```
bracket          : string -> string -> 'a parser -> 'a parser
scanToCloseParen : srcloc parser
```

```
fun bracket keyword expected p =
  "(" >-- literal keyword *> (p --< ")" <|>
                              errorAt ("expected " ^ expected) <$>! scanToCloseParen)
and scanToCloseParen tokens =
  let val loc = getOpt (peek srcloc tokens, ("end of stream", 9999))
      fun scan lpcount tokens =
        (* lpcount is the number of unmatched left parentheses *)
        case tokens
          of EOL _          ::: tokens => scan lpcount tokens
           | INLINE (_, t) ::: tokens =>
                                if isLiteral "(" t then scan (lpcount+1) tokens
                                else if isLiteral ")" t then
                                    if lpcount = 0 then SOME (OK loc, tokens)
                                    else scan (lpcount-1) tokens
                                else scan lpcount tokens
           | EOS          => SOME (errorAt "unmatched (" loc, EOS)
           | SUSPENDED s => scan lpcount (force s)
  in  scan 0 tokens
  end
```


--< 664c
::: 647a
<$>! 658c
<|> 654b
>-- 664c
EOL 662a
EOS 647a
errorAt 661b
force 646
INLINE 662a
isLiteral, in Typed μScheme 671b
in μProlog 710d
in μSmalltalk 699d
literal 664b
OK 651a
peek 655d
srcloc 663a
SUSPENDED 647a


[9] I have spent entirely too much time working with Englishmen who call parentheses "brackets." I now find it hard even to *say* the word "parenthesis," let alone put them in my code. So the function is called `bracket`.

### Detection of duplicate names

Most of the languages in this book allow you to define functions or methods that take formal parameters. It is never permissible to use the same name for formal parameters in two different positions. There are surprisingly many other places where it's not acceptable to have duplicate in a list of strings. Function **nodups** takes two Curried arguments: a pair saying what kind of thing might be duplicated and where it appeared, followed by a pair containing a list of names and the source-code location of the list. If there are no duplicates, it returns the list of names; otherwise it raises the `SyntaxError` exception.

666a ⟨*parsing utilities* 662b⟩+≡ (644) ◁665

```
nodups : string * string -> srcloc * name list -> name list error
fun nodups (what, where') (loc, names) =
  let fun dup [] = OK names
        | dup (x::xs) = if List.exists (fn y : string => y = x) xs then
                          errorAt (what ^ " " ^ x ^ " appears twice in " ^ where') loc
                        else
                          dup xs
  in  dup names
  end
```

Function `List.exists` is like the $\mu$Scheme `exists?`. It is in the initial basis for Standard ML.

## D.5.6 Code used to debug parsers

When debugging parsers, I often find it helpful to dump out the tokens that a parser is looking at. I want to dump all the tokens that are available *without* triggering the action of reading another line of input. I believe it's safe to read until I have got to *both* an end-of-line marker *and* an unforced suspension.

666b ⟨*code used to debug parsers* 666b⟩≡ (644) 667a▷

```
safeTokens : token located inline stream -> token list
val safeTokens : token located inline stream -> token list =
  let fun tokens (seenEol, seenUnforced) =
            let fun get (EOL _         ::: ts) = if seenUnforced then []
                                                 else tokens (true, false) ts
                  | get (INLINE (_, t) ::: ts) = t :: get ts
                  | get  EOS                   = []
                  | get (SUSPENDED (ref (PRODUCED ts))) = get ts
                  | get (SUSPENDED s) = if seenEol then []
                                        else tokens (false, true) (force s)
            in   get
            end
  in  tokens (false, false)
  end
```

| | |
|---|---|
| ::: | 647a |
| EOL | 662a |
| EOS | 647a |
| errorAt | 661b |
| force | 646 |
| INLINE | 662a |
| OK | 651a |
| PRODUCED | 646 |
| type stream | 647a |
| SUSPENDED | 647a |

The wrap function can be used to wrap a parser; it shows what the parser was looking for, what tokens it was looking at, and whether it found something.

667a ⟨*code used to debug parsers* 666b⟩+≡ (644) ◁666b

wrap : string -> 'a parser -> 'a parser

```
fun wrap what p tokens =
  let fun t tok = " " ^ tokenString tok
      val _ = app print ["Looking for ", what, " at"]
      val _ = app (print o t) (safeTokens tokens)
      val _ = print "\n"
      val answer = p tokens
      val _ = app print [case answer of NONE => "Didn't find " | SOME _ => "Found ",
                         what, "\n"]
  in  answer
  end handle e => ( app print ["Search for ", what, " raised ", exnName e, "\n"]
                  ; raise e)

fun wrap what p = p
```

## D.6 An interactive reader

In this final section I pull together all the machinery needed to take a stream of input lines, a lexer, and a parser, and to produce a stream of high-level syntactic objects like definitions. There's more to it than just stitching together functions of the right types; this code is where prompts get determined, where errors are handled, and where special tagged lines are copied to the output to support testing.

### Testing support

Let's get the testing support out of the way first. As in the C code, I want to print out any line read that begins with the special string ;#. This string is a formal comment that helps me test the chunks marked ⟨*transcript* (generated automatically)⟩. In the ML code, I can do the job in a very modular way: I define a post-stream action that prints any line meeting the criterion. Function echoTagStream transforms a stream of lines to a stream of lines, adding the behavior I want.

667b ⟨*an interactive reader* 667b⟩≡ (644) 667c▷

echoTagStream : line stream -> line stream

```
fun echoTagStream lines =
  let fun echoIfTagged line =
        if (String.substring (line, 0, 2) = ";#" handle _ => false) then
          print line
        else
          ()
  in  postStream (lines, echoIfTagged)
  end
```

postStream 649b
safeTokens 666b
tokenString, in Typed $\mu$Scheme 671b
in $\mu$Prolog 710c
in $\mu$Smalltalk 699c

### Lexing and parsing with error handling

The next step is error handling. When the code detects an error it prints a message using function errorln.

667c ⟨*an interactive reader* 667b⟩+≡ (644) ◁667b 668a▷

errorln : string -> unit

```
fun errorln s = TextIO.output (TextIO.stdErr, s ^ "\n")
```

The basic error handler strips and prints errors.

668a ⟨*an interactive reader* 667b⟩+≡ (644) ◁667c 668b▷

```
fun stripErrors xs =
  let fun next xs =
        case streamGet xs
          of SOME (ERROR msg, xs) => (errorln ("error: " ^ msg); next xs)
           | SOME (OK x, xs) => SOME (x, xs)
           | NONE => NONE
  in  streamOfUnfold next xs
  end
```

stripErrors : 'a error stream -> 'a stream

An error detected during lexical analysis is printed without any information about source-code locations. That's because, to keep things somewhat simple, I've chosen to do lexical analysis on one line at a time, and I don't keep track of the line's source-code location.

668b ⟨*an interactive reader* 667b⟩+≡ (644) ◁668a 668c▷

```
fun lexLineWith lexer =
  stripErrors o streamOfUnfold lexer o streamOfList o explode
```

lexLineWith : token lexer -> line -> token stream

When an error occurs during parsing, I drain the rest of the tokens on the line where the error occurred. I *don't* strip the errors at this point; errors need to make it through to the reader because when an error is detected, the prompt may need to be adjusted.

668c ⟨*an interactive reader* 667b⟩+≡ (644) ◁668b 668d▷

parseWithErrors : 'a parser -> token located inline stream -> 'a error stream

```
fun parseWithErrors parser =
  let fun adjust (SOME (ERROR msg, tokens)) = SOME (ERROR msg, drainLine tokens)
        | adjust other = other
  in  streamOfUnfold (adjust o parser)
  end
```

### Prompts

START HERE! WE BUILD ON THE UNIX SHELL MODEL OF HAVING TWO PROMPT STRINGS.

PS1 IS THE PROMPT THAT'S PRINTED WHEN THE INTERPRETER IS WAITING FOR THE USER TO ENTER A DEFINITION. PS2 IS THE PROMPT THAT'S PRINTED WHEN THE USER HITS A NEWLINE BUT THE CURRENT DEFINITION IS NOT YET COMPLETE. TO TURN OF PROMPTING ENTIRELY, SET THEM BOTH TO THE EMPTY STRING.

drainLine 662a
ERROR 651a
errorln 667c
OK 651a
streamGet 647b
streamOfList 647c
streamOfUnfold 648d

⟨*an interactive reader* 667b⟩+≡ (644) ◁668c 669▷

```
type prompts   = { ps1 : string, ps2 : string }
val stdPrompts = { ps1 = "-> ", ps2 = "    " }
val noPrompts  = { ps1 = "", ps2 = "" }
```

type prompts
stdPrompts : prompts
noPrompts : prompts

### Building a reader

The `reader` function has two jobs which are impossible to separate: to manage the flow of information from the input through the lexer and parser, and to arrange that the right prompts are printed at the right times. The flow of information involves multiple steps:

1. We start with a stream of lines. The stream is transformed with `preStream` and `echoTagStream`, so that a prompt is printed before every line, and when a line contains the special tag, that line is echoed to the output.

2. Function `lexLineWith lexer` converts a line to a stream of tokens, which then have to be paired with source-code locations, tagged with `INLINE`, and followed by an `EOL` value. This extra decoration gets us from the `token stream` provided by the lexer to the `token located inline stream` needed by the parser. The work is done by function `lexAndDecorate`, which needs a *located* line.

3. The final stream of definitions is computed by composing `locatedStream` to add source-code locations, `streamConcatMap lexAndDecorate` to add decorations, and `parseWithErrors` to parse. The entire composition is applied to the stream of lines created in step 1.

The other job of the `reader` function is to deliver the right prompt in the right situation. START EDITING HERE.

The prompt is initially `ps1`. It is set to `ps2` every time a token is produced, then reset to `ps1` every time we attempt to parse a definition.

669 ⟨*an interactive reader* 667b⟩+≡ (644) ◁668d

```
reader : token lexer * 'a parser -> prompts -> string * line stream -> 'a stream
lexAndDecorate : srcloc * line -> token located inline stream
```

```
fun 'a reader (lexer, parser) prompts (name, lines) =
  let val { ps1, ps2 } = prompts
      val thePrompt = ref ps1
      fun setPrompt ps = fn _ => thePrompt := ps

      val lines = preStream (fn () => print (!thePrompt), echoTagStream lines)

      fun lexAndDecorate (loc, line) =
        let val tokens = postStream (lexLineWith lexer line, setPrompt ps2)
        in  streamMap INLINE (streamZip (streamRepeat loc, tokens)) @@@
            streamOfList [EOL (snd loc)]
        end

      val edefs : 'a error stream =
        (parseWithErrors parser o streamConcatMap lexAndDecorate o locatedStream)
        (name, lines)
  in
      stripErrors (preStream (setPrompt ps1, edefs))
  end
```

## D.7 Further reading

Fat book by Aho and Ullman (1972).
Really nice paper by Knuth (1965).
Wirth (1977b) master of the hand-written recursive-descent parser.
Gibbons and Jones (1998)
Ramsey (1999)
(Mcbride and Paterson 2008)