Chapter 10

Streams and parsers

In the next part of these course notes, we will implement a small functional language. Parsing valid programs of this language requires writing a lexical analyzer and a parser for the language. For the purpose of writing easily such programs, Caml Light provides a special data structure: *streams*. Their main usage is to be interfaced to input channels or strings and to be matched against *stream patterns*.

10.1 Streams

Streams belong to an abstract data type: their actual representation remains hidden from the user. However, it is still possible to build streams either "by hand" or by using some predefined functions.

10.1.1 The `stream` type

The type `stream` is a parameterized type. One can build streams of integers, of characters or of any other type. Streams receive a special syntax, looking like the one for lists. The empty stream is written:

```
#[< >];;
- : '_a stream = <abstr>
```

A non empty stream possesses elements that are written preceded by the "'" (quote) character.

```
#[< '0; '1; '2 >];;
- : int stream = <abstr>
```

Elements that are not preceded by "'" are *substreams* that are expanded in the enclosing stream:

```
#[< '0; [<'1;'2>]; '3 >];;
- : int stream = <abstr>
#let s = [< '"abc" >] in [< s; '"def" >];;
- : string stream = <abstr>
```

Thus, stream concatenation can be defined as:

```
#let stream_concat s t = [< s; t >];;
stream_concat : 'a stream -> 'a stream -> 'a stream = <fun>
```

Building streams in this way can be useful while testing a parsing function or defining a lexical analyzer (taking as argument a stream of characters and returning a stream of tokens). Stream concatenation *does not copy* substreams: they are simply put in the same stream. Since (as we will see later) stream matching has a destructive effect on streams (streams are physically "eaten" by stream matching), parsing `[< t; t >]` will in fact parse `t` only once: the first occurrence of `t` will be consumed, and the second occurrence will be empty before its parsing will be performed.

Interfacing streams with an input channel can be done with the function:

```
#stream_of_channel;;
- : in_channel -> char stream = <fun>
```

returning a stream of characters which are read from the channel argument. The end of stream will coincide with the end of the file associated to the channel.

In the same way, one can build the character stream associated to a character string using:

```
#stream_of_string;;
- : string -> char stream = <fun>

#let s = stream_of_string "abc";;
s : char stream = <abstr>
```

10.1.2 Streams are lazily evaluated

Stream expressions are submitted to *lazy evaluation*, i.e. they are effectively build only when required. This is useful in that it allows for the easy manipulation of "interactive" streams like the stream built from the standard input. If this was not the case, i.e. if streams were immediately completely computed, a program evaluating "`stream_of_channel std_in`" would read everything up to an end-of-file on standard input before giving control to the rest of the program. Furthermore, lazy evaluation of streams allows for the manipulation of infinite streams. As an example, we can build the infinite stream of integers, using side effects to show precisely when computations occur:

```
#let rec ints_from n =
#   [< '(print_int n; print_char ' '; flush std_out; n);
#      ints_from (n+1) >];;
ints_from : int -> int stream = <fun>

#let ints = ints_from 0;;
ints : int stream = <abstr>
```

We notice that no printing occurred and that the program terminates: this shows that none of the elements have been evaluated and that the infinite stream has not been built. We will see in the next section that these side-effects will occur on demand, i.e. when tests will be needed by a matching function on streams.

10.2 Stream matching and parsers

The syntax for building streams can be used for pattern-matching over them. However, stream matching is more complex than the usual pattern matching.

10.2.1 Stream matching is destructive

Let us start with a simple example:

```
#let next = function [< 'x >] -> x;;
next : 'a stream -> 'a = <fun>
```

The `next` function returns the first element of its stream argument, and fails if the stream is empty:

```
#let s = [< '0; '1; '2 >];;
s : int stream = <abstr>

#next s;;
- : int = 0

#next s;;
- : int = 1

#next s;;
- : int = 2

#next s;;
Uncaught exception: Parse_failure
```

We can see from the previous examples that the stream pattern `[< 'x >]` matches *an initial segment* of the stream. Such a pattern must be read as "the stream whose first element matches `x`". Furthermore, once stream matching has succeeded, the stream argument has been *physically modified* and does not contain any longer the part that has been recognized by the `next` function.

If we come back to the infinite stream of integers, we can see that the calls to `next` provoke the evaluation of the necessary part of the stream:

```
#next ints; next ints; next ints;;
0 1 2 - : int = 2
```

Thus, successive calls to `next` remove the first elements of the stream until it becomes empty. Then, `next` fails when applied to the empty stream, since, in the definition of `next`, there is no stream pattern that matches an initial segment of the empty stream.

It is of course possible to specify several stream patterns as in:

```
#let next = function
#  [< 'x >] -> x
#| [< >] -> raise (Failure "empty");;
next : 'a stream -> 'a = <fun>
```

Cases are tried in turn, from top to bottom.

Stream pattern components are not restricted to quoted patterns (intended to match stream elements), but can be also function calls (corresponding to non-terminals, in the grammar terminology). Functions appearing as stream pattern components are intended to match substreams of the stream argument: they are called on the actual stream argument, and they are followed by a pattern which should match the result of this call. For example, if we define a parser recognizing a non empty sequence of characters `'a'`:

```
#let seq_a =
#    let rec seq = function
#        [< ''a'; seq l >] -> 'a'::l
#      | [< >] -> []
#    in function [< ''a'; seq l >] -> 'a'::l;;
seq_a : char stream -> char list = <fun>
```

we used the recursively defined function `seq` inside the stream pattern of the first rule. This definition should be read as:

- if the stream is not empty and if its first element matches `'a'`, apply `seq` to the rest of the stream, let `l` be the result of this call and return `'a'::l`,
- otherwise, fail (raise `Parse_failure`);

and `seq` should be read in the same way (except that, since it recognizes possibly empty sequences of `'a'`, it never fails).

Less operationally, we can read it as: "a non-empty sequence of `'a'` starts with an `'a'`, and is followed by a possibly empty sequence of `'a'`.

Another example is the recognition of a non-empty sequence of `'a'` followed by a `'b'`, or a `'b'` alone:

```
#let seq_a_b = function
#  [< seq_a l; ''b' >] -> l@['b']
#| [< ''b' >] -> ['b'];;
seq_a_b : char stream -> char list = <fun>
```

Here, operationally, once an `'a'` has been recognized, the first matching rule is chosen. Any further mismatch (either from `seq_a` or from the last `'b'`) will raise a `Parse_error` exception, and the whole parsing will fail. On the other hand, if the first character is not an `'a'`, `seq_a` will raise `Parse_failure`, and the second rule (`[< ''b' >] -> ...`) will be tried.

This behavior is typical of predictive parsers. Predictive parsing is recursive-descent parsing with one look-ahead token. In other words, a predictive parser is a set of (possibly mutually recursive) procedures, which are selected according to the shape of (at most) the first token.

10.2.2 Sequential binding in stream patterns

Bindings in stream patterns occur sequentially, in contrast with bindings in regular patterns, which can be thought as occurring in parallel. Stream matching is guaranteed to be performed from left to right. For example, computing the sum of the elements of an integer stream could be defined as:

```
#let rec stream_sum n = function
#  [< '0; (stream_sum n) p >] -> p
#| [< 'x; (stream_sum (n+x)) p >] -> p
#| [< >] -> n;;
stream_sum : int -> int stream -> int = <fun>

#stream_sum 0 [< '0; '1; '2; '3; '4 >];;
- : int = 10
```

The `stream_sum` function uses its first argument as an accumulator holding the sum computed so far. The call `(stream_sum (n+x))` uses `x` which was bound in the stream pattern component occurring at the left of the call.

Warning: streams patterns are legal only in the `function` and `match` constructs. The `let` and other forms are restricted to usual patterns. Furthermore, a stream pattern cannot appear inside another pattern.

10.3 Parameterized parsers

Since a parser is a function like any other function, it can be parameterized or be used as a parameter. Parameters used only in the right-hand side of stream-matching rules simulate *inherited attributes* of attribute grammars. Parameters used as parsers in stream patterns allow for the implementation of *higher-order* parsers. We will use the next example to motivate the introduction of parameterized parsers.

10.3.1 Example: a parser for arithmetic expressions

Before building a parser for arithmetic expressions, we need a lexical analyzer able to recognize arithmetic operations and integer constants. Let us first define a type for tokens:

```
#type token =
#  PLUS | MINUS | TIMES | DIV | LPAR | RPAR
#| INT of int;;
Type token defined.
```

Skipping blank spaces is performed by the `spaces` function defined as:

```
#let rec spaces = function
#  [< ' ' '|'\t'|'\n'; spaces _ >] -> ()
#| [< >] -> ();;
spaces : char stream -> unit = <fun>
```

The conversion of a digit (character) into its integer value is done by:

```
#let int_of_digit = function
#  '0'..'9' as c -> (int_of_char c) - (int_of_char '0')
#| _ -> raise (Failure "not a digit");;
int_of_digit : char -> int = <fun>
```

The "as" keyword allows for naming a pattern: in this case, the variable c will be bound to the actual digit matched by '0'..'9'. Pattern built with as are called *alias patterns*.

For the recognition of integers, we already feel the need for a parameterized parser. Integer recognition is done by the integer analyzer defined below. It is parameterized by a numeric value representing the value of the first digits of the number:

```
#let rec integer n = function
#  [< ' '0'..'9' as c; (integer (10*n + int_of_digit c)) r >] -> r
#| [< >] -> n;;
integer : int -> char stream -> int = <fun>

#integer 0 (stream_of_string "12345");;
- : int = 12345
```

We are now ready to write the lexical analyzer, taking a stream of characters, and returning a stream of tokens. Returning a token stream which will be explored by the parser is a simple, reasonably efficient and intuitive way of composing a lexical analyzer and a parser.

```
#let rec lexer s = match s with
#  [< ''('; spaces _ >] -> [< 'LPAR; lexer s >]
#| [< ')'; spaces _ >] -> [< 'RPAR; lexer s >]
#| [< '+'; spaces _ >] -> [< 'PLUS; lexer s >]
#| [< '-'; spaces _ >] -> [< 'MINUS; lexer s >]
#| [< '*'; spaces _ >] -> [< 'TIMES; lexer s >]
#| [< '/'; spaces _ >] -> [< 'DIV; lexer s >]
#| [< '0'..'9' as c; (integer (int_of_digit c)) n; spaces _ >]
#                       -> [< 'INT n; lexer s >];;
lexer : char stream -> token stream = <fun>
```

We assume there is no leading space in the input.

Now, let us examine the language that we want to recognize. We shall have integers, infix arithmetic operations and parenthesized expressions. The BNF form of the grammar is:

```
Expr ::= Expr + Expr
       | Expr - Expr
       | Expr * Expr
       | Expr / Expr
       | ( Expr )
       | INT
```

The values computed by the parser will be *abstract syntax trees* (by contrast with *concrete syntax*, which is the input string or stream). Such trees belong to the following type:

```
#type atree =
#  Int of int
#| Plus of atree * atree
#| Minus of atree * atree
#| Mult of atree * atree
#| Div of atree * atree;;
Type atree defined.
```

The `Expr` grammar is ambiguous. To make it unambiguous, we will adopt the usual precedences for arithmetic operators and assume that all operators associate to the left. Now, to use stream matching for parsing, we must take into account the fact that matching rules are chosen according to the behavior of the first component of each matching rule. This is predictive parsing, and, using well-known techniques, it is easy to rewrite the grammar above in such a way that writing the corresponding predictive parser becomes trivial. These techniques are described in [2], and consist in adding a non-terminal for each precedence level, and removing left-recursion. We obtain:

```
Expr ::= Mult
       | Mult + Expr
       | Mult - Expr

Mult ::= Atom
       | Atom * Mult
       | Atom / Mult

Atom ::= INT
       | ( Expr )
```

While removing left-recursion, we forgot about left associativity of operators. This is not a problem, as long as we build correct abstract trees.

Since stream matching bases its choices on the first component of stream patterns, we cannot see the grammar above as a parser. We need a further transformation, factoring common prefixes of grammar rules (left-factor). We obtain:

```
Expr ::= Mult RestExpr

        RestExpr ::= + Mult RestExpr
                   | - Mult RestExpr
                   | (* nothing *)

Mult ::= Atom RestMult

        RestMult ::= * Atom RestMult
                   | / Atom RestMult
                   | (* nothing *)

Atom ::= INT
       | ( Expr )
```

Now, we can see this grammar as a parser (note that the order of rules becomes important, and empty productions must appear last). The shape of the parser is:

```
let rec expr =
    let rec restexpr ? = function
        [< 'PLUS; mult ?; restexpr ? >] -> ?
      | [< 'MINUS; mult ?; restexpr ? >] -> ?
```

```
      | [< >] -> ?
in function [< mult e1; restexpr ? >] -> ?

and mult =
    let rec restmult ? = function
        [< 'TIMES; atom ?; restmult ? >] -> ?
      | [< 'DIV; atom ?; restmult ? >] -> ?
      | [< >] -> ?
in function [< atom e1; restmult ? >] -> ?

and atom = function
  [< 'INT n >] -> Int n
| [< 'LPAR; expr e; 'RPAR >] -> e
```

We used question marks where parameters, bindings and results still have to appear. Let us consider the `expr` function: clearly, as soon as `e1` is recognized, we must be ready to build the leftmost subtree of the result. This leftmost subtree is either restricted to `e1` itself, in case `restexpr` does not encounter any operator, or it is the tree representing the addition (or subtraction) of `e1` and the expression immediately following the additive operator. Therefore, `restexpr` must be called with `e1` as an intermediate result, and accumulate subtrees built from its intermediate result, the tree constructor corresponding to the operator and the last expression encountered. The body of `expr` becomes:

```
let rec expr =
    let rec restexpr e1 = function
        [< 'PLUS; mult e2; restexpr (Plus (e1,e2)) e >] -> e
      | [< 'MINUS; mult e2; restexpr (Minus (e1,e2)) e >] -> e
      | [< >] -> e1
in function [< mult e1; (restexpr e1) e2 >] -> e2
```

Now, `expr` recognizes a product `e1` (by `mult`), and applies `(restexpr e1)` to the rest of the stream. According to the additive operator encountered (if any), this function will apply `mult` which will return some `e2`. Then the process continues with `Plus(e1,e2)` as intermediate result. In the end, a correctly balanced tree will be produced (using the last rule of `restexpr`).

With the same considerations on `mult` and `restmult`, we can complete the parser, obtaining:

```
#let rec expr =
#    let rec restexpr e1 = function
#        [< 'PLUS; mult e2; (restexpr (Plus (e1,e2))) e >] -> e
#      | [< 'MINUS; mult e2; (restexpr (Minus (e1,e2))) e >] -> e
#      | [< >] -> e1
#in function [< mult e1; (restexpr e1) e2 >] -> e2
#
#and mult =
#    let rec restmult e1 = function
#        [< 'TIMES; atom e2; (restmult (Mult (e1,e2))) e >] -> e
#      | [< 'DIV; atom e2; (restmult (Div (e1,e2))) e >] -> e
```

```
#      | [< >] -> e1
#in function [< atom e1; (restmult e1) e2 >] -> e2
#
#and atom = function
#  [< 'INT n >] -> Int n
#| [< 'LPAR; expr e; 'RPAR >] -> e;;
expr : token stream -> atree = <fun>
mult : token stream -> atree = <fun>
atom : token stream -> atree = <fun>
```

And we can now try our parser:

```
#expr (lexer (stream_of_string "(1+2+3*4)-567"));;
- : atree = Minus (Plus (Plus (Int 1, Int 2), Mult (Int 3, Int 4)), Int 567)
```

10.3.2 Parameters simulating inherited attributes

In the previous example, the parsers `restexpr` and `restmult` take an abstract syntax tree `e1` as argument and pass it down to the result through recursive calls such as `(restexpr (Plus(e1,e2)))`. If we see such parsers as non-terminals (`RestExpr` from the grammar above) this parameter acts as an inherited attribute of the non-terminal. Synthesized attributes are simulated by the right hand sides of stream matching rules.

10.3.3 Higher-order parsers

In the definition of `expr`, we may notice that the parsers `expr` and `mult` on the one hand and `restexpr` and `restmult` on the other hand have exactly the same structure. To emphasize this similarity, if we define parsers for additive (resp. multiplicative) operators by:

```
#let addop = function
#  [< 'PLUS >] -> (function (x,y) -> Plus(x,y))
#| [< 'MINUS >] -> (function (x,y) -> Minus(x,y))
#and multop = function
#  [< 'TIMES >] -> (function (x,y) -> Mult(x,y))
#| [< 'DIV >] -> (function (x,y) -> Div(x,y));;
addop : token stream -> atree * atree -> atree = <fun>
multop : token stream -> atree * atree -> atree = <fun>
```

we can rewrite the `expr` parser as:

```
#let rec expr =
#    let rec restexpr e1 = function
#        [< addop f; mult e2; (restexpr (f (e1,e2))) e >] -> e
#      | [< >] -> e1
#in function [< mult e1; (restexpr e1) e2 >] -> e2
#
#and mult =
```

```
#    let rec restmult e1 = function
#        [< multop f; atom e2; (restmult (f (e1,e2))) e >] -> e
#      | [< >] -> e1
#in function [< atom e1; (restmult e1) e2 >] -> e2
#
#and atom = function
#  [< 'INT n >] -> Int n
#| [< 'LPAR; expr e; 'RPAR >] -> e;;
expr : token stream -> atree = <fun>
mult : token stream -> atree = <fun>
atom : token stream -> atree = <fun>
```

Now, we take advantage of these similarities in order to define a general parser for left-associative operators. Its name is `left_assoc` and is parameterized by a parser for operators and a parser for expressions:

```
#let rec left_assoc op term =
#    let rec rest e1 = function
#        [< op f; term e2; (rest (f (e1,e2))) e >] -> e
#      | [< >] -> e1
#    in function [< term e1; (rest e1) e2 >] -> e2;;
left_assoc :
 ('a stream -> 'b * 'b -> 'b) -> ('a stream -> 'b) -> 'a stream -> 'b = <fun>
```

Now, we can redefine `expr` as:

```
#let rec expr str = left_assoc addop mult str
#and mult str = left_assoc multop atom str
#and atom = function
#  [< 'INT n >] -> Int n
#| [< 'LPAR; expr e; 'RPAR >] -> e;;
expr : token stream -> atree = <fun>
mult : token stream -> atree = <fun>
atom : token stream -> atree = <fun>
```

And we can now try our definitive parser:

```
#expr (lexer (stream_of_string "(1+2+3*4)-567"));;
- : atree = Minus (Plus (Plus (Int 1, Int 2), Mult (Int 3, Int 4)), Int 567)
```

Parameterized parsers are useful for defining general parsers such as `left_assoc` that can be used with different instances. Another example of a useful general parser is the `star` parser defined as:

```
#let rec star p = function
#  [< p x; (star p) l >] -> x::l
#| [< >] -> [];;
star : ('a stream -> 'b) -> 'a stream -> 'b list = <fun>
```

The `star` parser iterates zero or more times its argument `p` and returns the list of results. We still have to be careful in using these general parsers because of the predictive nature of parsing. For example, `star p` will never successfully terminate if `p` has a rule for the empty stream pattern: this rule will make the second rule of `star` useless!

10.3.4 Example: parsing a non context-free language

As an example of parsing with parameterized parsers, we shall build a parser for the language $\{wCw \mid w \in (A|B)^*\}$, which is known to be non context-free.

First, let us define a type for this alphabet:

```
#type token = A | B | C;;
Type token defined.
```

Given an input of the form w`C`w, the idea for a parser recognizing it is:

- first, to recognize the sequence w with a parser `wd` (for *word definition*) returning information in order to build a parser recognizing only w;
- then to recognize `C`;
- and to use the parser built at the first step to recognize the sequence w.

The definition of `wd` is as follows:

```
#let rec wd = function
#  [< 'A; wd l >] -> (function [< 'A >] -> "a")::l
#| [< 'B; wd l >] -> (function [< 'B >] -> "b")::l
#| [< >] -> [];;
wd : token stream -> (token stream -> string) list = <fun>
```

The `wu` function (for *word usage*) builds a parser sequencing a list of parsers:

```
#let rec wu = function
#  p::pl -> (function [< p x; (wu pl) l >] -> x^l)
#| [] -> (function [< >] -> "");;
wu : ('a stream -> string) list -> 'a stream -> string = <fun>
```

The `wu` function builds, from a list of parsers p_i, for $i = 1..n$, a single parser

$$\texttt{function [<}p_1\ x_1\texttt{;}\ldots\texttt{;}p_n\ x_n\texttt{>] -> [}x_1\texttt{;}\ldots\texttt{;}x_n\texttt{]}$$

which is the sequencing of parsers p_i. The main parser `w` is:

```
#let w = function [< wd l; 'C; (wu l) r >] -> r;;
w : token stream -> string = <fun>

#w [< 'A; 'B; 'B; 'C; 'A; 'B; 'B >];;
- : string = "abb"

#w [< 'C >];;
- : string = ""
```

In the previous parser, we used an intermediate list of parsers in order to build the second parser. We can redefine `wd` without using such a list:

```
#let w =
#    let rec wd wr = function
#        [< 'A; (wd (function [< wr r; 'A >] -> r^"a")) p >] -> p
#      | [< 'B; (wd (function [< wr r; 'B >] -> r^"b")) p >] -> p
#      | [< >] -> wr
#    in function [< (wd (function [< >] -> "")) p; 'C; p str >] -> str;;
w : token stream -> string = <fun>
#w [< 'A; 'B; 'B; 'C; 'A; 'B; 'B >];;
- : string = "abb"
#w [< 'C >];;
- : string = ""
```

Here, `wd` is made local to `w`, and takes as parameter `wr` (for *word recognizer*) whose initial value is the parser with an empty stream pattern. This parameter accumulates intermediate results, and is delivered at the end of parsing the initial sequence w. After checking for the presence of `C`, it is used to parse the second sequence w.

10.4 Further reading

A summary of the constructs over streams and of primitives over streams is given in [20].

An alternative to parsing with streams and stream matching are the `camllex` and `camlyacc` programs.

A detailed presentation of streams and stream matching following "predictive parsing" semantics can be found in [25], where alternative semantics are given with some possible implementations.

Exercises

Exercise 10.1 *Define a parser for the language of prefix arithmetic expressions generated by the grammar:*

```
Expr ::= INT
       | + Expr Expr
       | - Expr Expr
       | * Expr Expr
       | / Expr Expr
```

Use the lexical analyzer for arithmetic expressions given above. The result of the parser must be the integer resulting from the evaluation of the arithmetic expression, i.e. its type must be:

```
token -> int
```

Exercise 10.2 *Enrich the type* `token` *above with a constructor* `IDENT of string` *for identifiers, and enrich the lexical analyzer for it to recognize identifiers built from alphabetic letters (upper or lowercase). Length of identifiers may be limited.*