

# Systematic Compiler Construction

Michael Sperber      Peter Thiemann

October 28, 2003

Dieses Werk ist urheberrechtlich geschützt; alle Rechte mit Ausnahme der folgenden sind vorbehalten:

*Studenten der Informatik dürfen diese Dokument für ihre persönlichen Lehrzwecke verwenden.*

Ausdrücklich verboten wird jede andere Verwendung von Ausdrucken, Dateikopien oder Paperkopien. Insbesondere darf dieses Skriptum nicht in irgendeiner Weise vertrieben werden und es dürfen keine Kopien der Dateien auf öffentlich zugänglichen Servern angelegt werden.

Copyright © Michael Sperber, 1999; Peter Thiemann, 2000, 2002

# Chapter 1

## Lexical Analysis

A small psychological exercise demonstrates what lexical analysis is. Read aloud the following Caml program:

```
let main () =  
  print_string "Caml is not an animal.\n"
```

Listening to yourself, you will notice the following peculiarities:

1. You probably read the program word by word (rather than letter by letter).
2. You may have treated (mentally or vocally) `()` as a unit rather than as two parentheses.
3. You did not read aloud spaces and line breaks.
4. You (mentally or vocally) treated the string `"Caml is not an animal.\n"` as a unit.

Most programming languages since the days of FORTRAN are structured in such a way that a program splits into a linear sequence of “words.” In the process of determining the boundaries between “words,” certain insignificant elements of a program (such as comments) drop out. Since no higher-level syntactic processing is involved in this phase, it makes sense to perform this splitting before further syntactic analysis. This splitting is called “lexical analysis.” The result of lexical analysis for the above program might look like this:

```
⟨let⟩  
⟨identifier [main]⟩  
⟨()⟩  
⟨=⟩  
⟨identifier [print_string]⟩  
⟨string [Caml is not an animal.↵]⟩
```

Some of the details of this splitting may seem arbitrary at this point: Why does `let` become a unit `⟨let⟩` while `main` is denoted as `⟨identifier [main]⟩`? The answer is this: the word `let` appears in the Caml language definition as one of the *keywords* of the language. Since the language definition is finite, the number of keywords must also be finite: Hence, it makes sense to treat keywords as special unit. The word `main`, on the other hand, does not appear in the language definition: `main` is an *identifier* chosen by the programmer. There are infinitely many identifiers, but they all share the same syntactic status: Hence the `⟨identifier [main]⟩` encoding which separates this information from the actual name of the identifier.

## 1.1 Basic definitions

The part of a compiler responsible for lexical analysis is called a *scanner* or a *lexer*. It operates on the program as a sequence of characters and performs three main tasks:

1. it divides the input into logically cohesive sequences of characters, the *lexemes*;
2. it filters out formatting characters, like spaces, tabulators, and newline characters (*whitespace* characters); and
3. it maps lexemes into *tokens*, *i.e.*, symbolic names for classes of lexemes. Most tokens carry *attributes* which are computed from the lexeme. There is a one-to-one correspondence between lexemes and token/attribute pairs.

### 1.1 Definition (Lexical analysis)

Let  $\Sigma$  be the alphabet of a programming language,  $T$  a finite set of tokens, and  $A$  an arbitrary set of attributes. A lexical analysis (or scanner or lexer) is a function

$$\text{scan} : \Sigma^* \rightarrow (T \times A)^*$$

such that there exists a function *unscan* with:

$$\text{unscan} : (T \times A)^* \rightarrow \Sigma^*.$$

*unscan* has the following properties:

$$\text{scan} \circ \text{unscan} = \text{id}_{(T \times A)^*} \tag{1}$$

There is a function  $\text{unscan}' : (T \times A) \rightarrow \Sigma^*$  with:

$$\text{unscan}(p_0 p_1 \dots p_n) = \text{unscan}'(p_0) \text{unscan}'(p_1) \dots \text{unscan}'(p_n) \tag{2}$$

□

Thus, the *scan* function removes parts of the input. The rest, it partitions into *lexemes*. A lexeme always corresponds to exactly one token/attribute pair. Consequently, the job of lexical analysis is to determine the lexemes, and assign a token/attribute pair to each of them. Language definitions use—by convention—regular expressions to define the lexemes of a programming language.

### 1 Example

The Caml reference manual contains a section “Lexical Conventions”. Figure 1.1 shows its description of the lexemes for identifiers and integer constants.

Lexical analysis exists for chiefly pragmatic reasons: the more involved syntactic analysis which follows can be much simpler because of it. Moreover, regular grammars have well-known algorithms to recognize them. Theoretical computer science tells us that a finite, deterministic automaton (DFA) can serve as a recognizer for any regular language. A DFA is a simple machine, and thus reasonably easy to implement. Unfortunately, the construction of the state diagram for a DFA is an involved and tedious process. Therefore, it makes things easier to try to circumvent the explicit construction of the automaton. Fortunately, the automaton follows automatically from our simpler approach to recognizing regular languages.

```

⟨ident⟩ ::= ⟨letter⟩ ⟨ident-rest⟩*
⟨letter⟩ ::= A | B | C | ... | Z | a | b | c | ... | z
⟨ident-rest⟩ ::= ⟨letter⟩ | ⟨digit⟩ | _ | '
⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨integer-literal⟩ ::= ⟨sign⟩ ⟨digit⟩+
                    | ⟨sign⟩ ⟨hexprefix⟩ ⟨hexdigit⟩+
                    | ⟨sign⟩ ⟨octprefix⟩ ⟨octdigit⟩+
                    | ⟨sign⟩ ⟨binprefix⟩ ⟨bindigit⟩+
⟨sign⟩ ::= ⟨empty⟩ | -
⟨empty⟩ ::=
⟨hexprefix⟩ ::= 0x | 0X
⟨hexdigit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
              | A | B | ... | F | a | b | ... | f
⟨octprefix⟩ ::= 0o | o0
⟨octdigit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
⟨binprefix⟩ ::= 0b | 0B
⟨bindigit⟩ ::= 0 | 1

```

Figure 1.1: Some lexical conventions of Caml

## 1.2 A module for regular expressions

This section deals with the implementation of regular expressions and the related algorithms in Caml. The following code belongs to a structured called **Regexp**.

A Caml declaration defines a data type for regular expressions, parameterized over the underlying alphabet of the language.

```

type 'a regexp =
  Null
  | Epsilon
  | Symbol of 'a
  | Concat of 'a regexp * 'a regexp
  | Alternate of 'a regexp * 'a regexp
  | Repeat of 'a regexp

```

The data type `'a regexp` can express the regular expression `Concat(x, Null)` which is equivalent to `Null`. Thus, the term language defined by `'a regexp` contains ambiguities. Specifically, it is possible to make due with regular expressions which contain no internal `Null` constructors; it is always possible to transform it to a regular expression which is either `Null` or does not contain `Null` at all. Therefore, it is a good idea to abstract over the constructors and perform some simplification on the way. Besides the elimination of internal `Null` constructors, the abstractions also get rid of some `Epsilon` constructors:

```

let epsilon = Epsilon
let symbol x = Symbol(x)
let concat r1 r2 =
  if r1 = Null or r2 = Null
  then Null
  else if r1 = Epsilon
  then r2
  else if r2 = Epsilon

```

```

    then r1
    else Concat(r1, r2)
let alternate r1 r2 =
  if r1 = Null
  then r2
  else if r2 = Null
  then r1
  else Alternate(r1, r2)
let repeat r =
  if r = Null or r = Epsilon
  then Epsilon
  else Repeat(r)

```

Some simple functions are useful in creating composite regular expressions:

```

let repeat_one r = concat r (repeat r)

let concat_list l = List.fold_left concat Epsilon l

let alternate_list l = List.fold_left alternate Null l

```

Now that there is functionality for *creating* regular expressions, the next job is to check, for a given sequence of alphabet symbols `symbols`, if it belongs to the language defined by a regular expression `regex`. The function `matches` will do exactly that. Its first few lines are straightforward. (It needs to prefix names from module `Regex` because it resides in a different module.)

```

let rec matches regex symbols =
  match symbols with
  [] -> Regex.accepts_empty regex
  | symbol::rest ->

```

This code calls a function `accepts_empty : 'a regex -> bool` that checks if the empty sequence belongs to the language of the regular expression. (The implementation of `accepts_empty` is a simple exercise.)

Now that the empty sequence is covered, non-empty sequences are next. This becomes easy in the presence of an auxiliary function `after_symbol`:

```

after_symbol : 'a -> 'a regex -> 'a regex

```

This function has the following behavior:

Let  $r$  be a regular expression describing the language  $L(r)$ . Let  $x\xi$  be a sequence of symbols. Then:

$$x\xi \in L(r) \iff \xi \in L(\text{after\_symbol } x \ r)$$

Thus, `after_symbol` “subtracts”  $x$  from  $r$ .

With `after_symbol`, `matches` is easy to complete:

```

    let next_regex = Regex.after_symbol symbol regex in
    if Regex.is_null next_regex
    then false
    else matches next_regex rest

```

Obviously, `matches` calls a simple function `is_null` from `Regex` with the following definition:

```

let is_null r = (r = Null)

```

If a regular expression was constructed exclusively by `epsilon`, `symbol`, `concat`, `alternate` and `repeat`, `is_null` tests reliably if a regular expression denotes the empty language.

The `Regexp.after_symbol` function is recursive over the construction of regular expressions:

```
let rec after_symbol symbol regexp =
  match regexp with
  | Null -> Null
  | Epsilon -> Null
  | Symbol(symbol') ->
    if symbol = symbol'
    then Epsilon
    else Null
  | Concat(r1, r2) ->
    let after_1 = concat (after_symbol symbol r1) r2 in
    let after_2 = if accepts_empty r1
                  then after_symbol symbol r2
                  else Null
    in
    alternate after_1 after_2
  | Alternate(r1, r2) ->
    alternate (after_symbol symbol r1)
              (after_symbol symbol r2)
  | Repeat(r1) ->
    concat (after_symbol symbol r1)
           (Repeat(r1))
```

(The proof for the correctness of `after_symbol` is a simple exercise.)

The `matches` function is *tail recursive*. Consequently, it implements a deterministic automaton with `after_symbol` as its state transition function. Compiler construction folklore constructs this automaton by first converting the regular expressions into a non-deterministic finite automaton and then making it deterministic. The approach presented here is much easier to implement and also easier to prove correct.

The `Regexp` structure is now complete. It has the following signature:

```
type 'a regexp

val epsilon : 'a regexp
val symbol : 'a -> 'a regexp
val concat : 'a regexp -> 'a regexp -> 'a regexp
val alternate : 'a regexp -> 'a regexp -> 'a regexp
val repeat : 'a regexp -> 'a regexp

val repeat_one : 'a regexp -> 'a regexp
val concat_list : 'a regexp list -> 'a regexp
val alternate_list : 'a regexp list -> 'a regexp

val is_null : 'a regexp -> bool

val accepts_empty : 'a regexp -> bool
val after_symbol : 'a -> 'a regexp -> 'a regexp
```

## 1.3 The construction of a scanner

The `matches` function of the previous section is not directly usable for lexical analysis: a scanner must recognize a number of different lexeme languages, it must consider a sequence of (potentially different) lexemes, and the scanner must turn each lexeme into a token/attribute pair. In addition, ambiguities can arise if the lexeme languages have overlaps. Hence, the description of a scanner comprises not just a single regular expression, but rather a whole bunch of them, together with instructions on how to turn the lexemes into token/attribute pairs.

### 1.3.1 Scanner descriptions

Instructions on how to turn lexemes into token/attribute pairs can be expressed as follows:

```
type ('a, 'token, 'attrib) lex_action =
  'a list * 'a list -> 'token * 'attrib * 'a list
```

In this declaration, `'a` is still the type of the alphabet of the language, `'token` is the type of the tokens, and `'attrib` is the type of the attributes. A `lex_action` assigns a function to a regular expression. Its parameter is a pair (*lexeme*, *rest*). The *lexeme* parameter denotes the lexeme which matches the regular expression, and *rest* is the rest of the input. The *rest* parameter is present because some `lex_action` functions (those that handle comments, for instance) must skip an initial part of *rest*. The function returns the token/attribute pair and the part of the input with which scanning can continue.

A rule in a scanner description simply pairs up a regular expression with a `lex_action`:

```
type ('a, 'token, 'attrib) lex_rule = 'a Regexp.regexp
  * ('a, 'token, 'attrib) lex_action
```

This completes yet another structure—`Lexspec`.

### 1.3.2 Scanner states

The job of a scanner is to successively consume symbols from the input, and, on recognizing a completed lexeme, to call the corresponding `lex_action`. To this end, the scanner must keep a state around which tracks which regular expressions may still match the part of the input consumed so far:

```
type ('a, 'token, 'attrib) lex_state =
  (('a, 'token, 'attrib) Lexspec.lex_rule) list
```

When the scanner consumes a symbol, it applies to all regular expressions of a `lex_state` the `Regexp.after_symbol` function (just like `matches`), and filters out the “dead” regular expressions that only match the empty language:

```
let next_state state symbol =
  let after_state =
    List.map (function (regexp, action) ->
      Regexp.after_symbol symbol regexp, action)
      state
  in
  filter (function (regexp, _) ->
    not (Regexp.is_null regexp))
    after_state
```

The eminently useful auxiliary function `filter` has the following—tail-recursive—definition:

```
let filter pred l =
  let rec f l r =
    match l with
    | [] -> List.rev r
    | x::xs ->
      if pred x
      then f xs (x::r)
      else f xs r
  in f l []
```

So a scanner description (a list of `lex_rules`) is easy to turn into an initial state for the scanner automaton:

```
let initial_state rules = rules
```

To determine which regular expressions have matched the consumed lexeme completely, the scanner uses the `matched_rules` function:

```
let matched_rules state =
  filter (function (regexp, _) ->
    Regexp.accepts_empty regexp)
    state
```

It is possible for the scanner to end up in a state where no further consumption of input symbols is possible. The `is_stuck` predicate diagnoses this situation:

```
let is_stuck state = state = []
```

This completes the `Lexstate` structure which has the following signature:

```
type ('a, 'token, 'attrib) lex_state

val next_state : ('a, 'token, 'attrib) lex_state
                -> 'a
                -> ('a, 'token, 'attrib) lex_state
val initial_state : (('a, 'token, 'attrib) Lexspec.lex_rule) list
                  -> ('a, 'token, 'attrib) lex_state
val matched_rules : ('a, 'token, 'attrib) lex_state
                  -> (('a, 'token, 'attrib) Lexspec.lex_rule) list
val is_stuck : ('a, 'token, 'attrib) lex_state -> bool
```

### 1.3.3 Resolution of ambiguities

Descriptions of lexical analysis for realistic programming languages almost always contain ambiguities: It is not sufficient to place the border between two lexemes arbitrarily as soon as some prefix has matched a regular expression. The fragments:

```
if n = 0 then 0 else n * fib(n-1)
```

```
and
```

```
ifoundsalvationinapubliclavatory
```

are syntactically correct Caml expressions starting with `if`. Now, the lexical syntax of Caml would allow to partition `ifoundsalvationinapubliclavatory` into lexemes in several different ways: either into `if` and `ifoundsalvationinapubliclavatory` or just just as `ifoundsalvationinapubliclavatory`. Obviously (or is it?), the latter alternative is the intended one.

Here is a quote from the Caml handbook that explains why:

Lexical ambiguities are resolved according to the “longest match” rule: when a character sequence can be decomposed into two tokens in several different ways, the decomposition retained is the one with the longest first token.

This longest match rule is part of the definitions of most programming languages. Therefore, most scanner generators use it by default.

### 1.3.4 Implementation of lexical analysis

All the building blocks for implementing lexical analysis are now in place. This section describes a Caml structure called `Lex` which contains the central functionality for creating scanners. The main function is called `scan_one`; it runs the state automaton in `Lexstate` to extract a single lexeme at the beginning of the input. To implement the “longest match” rule, `scan_one` remembers the last state in which it recognized a lexeme. Once `scan_one` has recognized a lexeme, it runs the corresponding action from the scanner description to yield a token/attribute pair and the rest of the input still to be processed.

```
exception Scan_error

let scan_one spec l =
  let rec loop state rev_lexeme maybe_last_match rest =
    if (Lexstate.is_stuck state) or (rest = [])
    then
      match maybe_last_match with
      | None -> raise Scan_error
      | Some last_match -> last_match
    else
      let symbol::rest = rest in
      let new_state = Lexstate.next_state state symbol in
      let new_matched = Lexstate.matched_rules new_state in
      let rev_lexeme = symbol::rev_lexeme in
      let maybe_last_match =
        match new_matched with
        | [] -> maybe_last_match
        | (_, action)::_ -> Some (action, rev_lexeme, rest)
      in
      loop new_state rev_lexeme maybe_last_match rest
  in
  let (action, rev_lexeme, rest) =
    loop (Lexstate.initial_state spec) [] None l
  in
  action (List.rev rev_lexeme, rest)
```

A scanner for a given programming language may consist of several parts, each with its own scanner description. The components may be composed via `let rec`:

```
let rec scan_1 input = Lex.scan_one <desc1> input
```

```
and scan_2 input = Lex.scan_one <desc2> input
and ...
```

Given a function which recognizes a single lexeme, it is easy to construct the complete scanner which turns the input—a list of symbols—into a list of token/attribute pairs:

```
let make_scanner scan_one input =
  let rec scan rev_result rest =
    if rest = []
    then List.rev rev_result
    else
      let (token, attrib, rest) = scan_one rest in
      scan ((token, attrib)::rev_result) rest
  in
  scan [] input
```