Systematic Compiler Construction

Michael Sperber Peter Thiemann

April 7, 2024

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 pecularities:

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 involed 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 Σ be the alphabet of a programming language, T a finite set of tokens, *and A an arbitrary set of* attributes. *A* tokenizer *is a function*

$$\mathit{tokenize} : \Sigma^* \to (T \times A)^*$$

such that there is a function

$$\mathit{untokenize} : (T \times A)^* \to \Sigma^*$$

with the following properties

1. $\mathit{untokenize} \circ \mathit{tokenize} = \mathit{id}_{(T \times A)^*}$ *and*
2. *there is a function* $\mathit{untoken} : (T \times A) \to \Sigma^*$ *so that*

$$\mathit{untokenize}(t_1 t_2 \dots) = \mathit{untoken}(t_1)\mathit{untoken}(t_2)\dots$$

*(*i.e., *untokenize is a homomorphism).*

If there is a distinguished token $WS \in T$ (whitespace) where $\mathit{untoken}(WS, a)$ is not a prefix of any other lexeme, then define

$$\begin{array}{lcl} \mathit{scan} & : & \Sigma^* \to (T \times A)^* \\ \mathit{unscan} & : & (T \times A)^* \to \Sigma^* \end{array}$$

as follows:

$$\begin{array}{lcl} \mathit{scan} & = & \mathit{filter}\ \mathit{notWS} \circ \mathit{tokenize} \\ \mathit{unscan} & = & \mathit{untokenize} \circ \mathit{insertWS} \\ \mathit{notWS}\ (WS, \_) & = & \mathit{false} \\ \mathit{notWS}\ (\_, \_) & = & \mathit{true} \\ \mathit{insertWS}\ [\,] & = & [\,] \\ \mathit{insertWS}\ (x :: xs) & = & x :: WS :: \mathit{insertWS}\ xs \end{array}$$

□

The functions *scan* and *unscan* have the following derived property.

$$\mathit{scan} \circ \mathit{unscan} \circ \mathit{scan} = \mathit{scan}$$

Thus, the *tokenize* function splits up the input into lexemes and maps them to token/attribute pairs. This process is clearly reversible. The *scan* function additionally removes those parts of the input which are labeled as whitespace.

1.2 Construction of scanners

There is a spectrum of possibilities for constructing scanners. First, a scanner can be implemented manually. A manual implementation can be tuned for efficiency and it does not place any restrictions on the language of lexemes. While this approach might be sensible for a simple language with few classes of lexemes, it is not appropriate for rich modern languages. Furthermore, a hand-written scanner is hard to maintain because it must be implemented from a separate specification. Last, the efficient implementation of some components of a scanner (*e.g.*, input buffering) is a non-trivial effort. A hand-written scanner must develop these components from scratch and cannot reuse previous efforts.

A second possibility is to rely on a prefabricated library of scanner components. In this approach, the implementation of the scanner can be close to the specification. Thus, it is amenable to fast development and easy maintainance. The downside is that the library cannot easily take advantage of global optimizations, so that such an implementation is significantly slower. Also, there are usually restrictions on the language of lexemes.

The third possibility is using a scanner generator like `lex` or `ocamllex`. A scanner generator generates the implementation of the scanner from a high-level specification. Thus, it combines the advantages of the library approach with efficiency. As with the library approach, the language of lexemes is usually restricted.

For instructional purposes, we concentrate on the second possibility. While sacrificing some efficiency, the library approach enables us to discuss the entire implementation of a scanner, without having to gloss over details of code generation as it would be the case with a scanner generator.

Both, the library approach and the generator approach, restrict the language of lexemes to a *regular language* R, for a number of reasons.

- All set-theoretic operations (union, intersection, difference) on regular languages yield regular languages. Hence, a specification of a regular language can rely on them.
- The word problem $w \in R$ is decidable in linear time in the length of the word w.
- For each regular language there is a minimal recognizer. Hence, there exists an optimal scanner for each kind of lexeme.

1.3 Descriptions of regular languages

Theoretical computer science tells us that there are at least three equivalent means of describing regular languages. We briefly introduce each of them and conclude with a discussion of which description is best suitable for specifying and implementing a scanner.

1.3.1 Regular grammars

Each regular language may be described by a regular grammar. A regular grammar is a grammar $\mathcal{G} = (N, \Sigma, P, S)$ where each production in P has one of the following forms:

$$A \to x\,B, \qquad A \to x, \qquad A \to \varepsilon$$

where $A, B \in N$, $x \in \Sigma$, and ε is the empty word. A word w belongs to the language defined by $\mathcal{G}$ iff there is a derivation $S \overset{*}{\Rightarrow} w$.

1.3.2 Finite automata

Another description of a regular language R is a finite automaton that recognizes R. A finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ consists of a finite set of states, Q, a finite input alphabet, Σ, a transition operator, δ, an initial state, $q_0 \in Q$, and a set of final states $F \subseteq Q$. Finite automata come in different flavors of equal power, distinguished by the definition of δ. The basic idea, however, is the same: at any time, the automaton is in a state $q \in Q$ and δ yields a new state from the current state and an input symbol. A word belongs to the language $L(M)$ recognized by M iff the automaton is in a final state after consuming all input symbols.

Deterministic finite automata (DFA)

A finite automaton is deterministic if $\delta : Q \times \Sigma \to Q$ is a function. To define $L(M)$, we extend δ to a function $\widehat{\delta} : Q \times \Sigma^* \to Q$ as follows:

$$\begin{array}{lcl} \widehat{\delta}(q, \varepsilon) & = & q \\ \widehat{\delta}(q, a\,w) & = & \widehat{\delta}(\delta(q, a), w) \end{array}$$

With this definition, $w \in L(M)$ iff $\widehat{\delta}(q_0, w) \in F$.

Nondeterministic finite automaton (NFA)

A finite automaton is non-deterministic if $\delta \subseteq Q \times \Sigma \times Q$ is an arbitrary relation. Again, to define $L(M)$, we extend δ to a relation $\widehat{\delta} \subseteq Q \times \Sigma^* \times Q$, which is the least relation satisfying the following two equations:

$$\begin{array}{l} (q, \varepsilon, q) \in \widehat{\delta} \\ (q, a\,w, q') \in \widehat{\delta} \quad \text{iff} \quad (\exists q'')\ (q, a, q'') \in \delta \text{ and } (q'', w, q') \in \widehat{\delta} \end{array}$$

With this definition, $w \in L(M)$ iff $(\exists q_f \in F)\ (q_0, w, q_f) \in \widehat{\delta}$.

Nondeterministic finite automata with autonomous transitions (NFA-ε)

An NFA may also allow autonomous (or instantaneous or ε) transitions which change the state without consuming any input. In this case, $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is an arbitrary relation. Again, we extend δ to the least relation $\widehat{\delta} \subseteq Q \times \Sigma^* \times Q$ which satisfies the following equations:

$$\begin{array}{lll} (q, \varepsilon, q) \in \widehat{\delta} & & \\ (q, w, q') \in \widehat{\delta} & \text{iff} & (\exists q'')\ (q, \varepsilon, q'') \in \delta \text{ and } (q'', w, q') \in \widehat{\delta} \\ (q, a\,w, q') \in \widehat{\delta} & \text{iff} & (\exists q'')\ (q, a, q'') \in \delta \text{ and } (q'', w, q') \in \widehat{\delta} \end{array}$$

With this definition, $w \in L(M)$ iff $(\exists q_f \in F)\ (q_0, w, q_f) \in \widehat{\delta}$.

1.3.3 Regular expressions

A regular expression is a highly declarative way of specifying a regular language. The set of regular expressions over an alphabet Σ is the smallest set $RE(\Sigma)$ with:

- $\underline{\emptyset} \in RE(\Sigma)$
- $\underline{\varepsilon} \in RE(\Sigma)$
- if $a \in \Sigma$ then $\underline{\mathtt{a}} \in RE(\Sigma)$
- if $r_1, r_2 \in RE(\Sigma)$ then $r_1 r_2 \in RE(\Sigma)$

- if $r_1, r_2 \in RE(\Sigma)$ then $r_1 \mid r_2 \in RE(\Sigma)$
- if $r \in RE(\Sigma)$ then $r^* \in RE(\Sigma)$.

A regular expression defines a language as prescribed by the function $L : RE(\Sigma) \to \mathcal{P}(\sigma^*)$.

$$\begin{array}{lcl} L(\underline{\emptyset}) & = & \emptyset \\ L(\underline{\varepsilon}) & = & \{\varepsilon\} \\ L(\underline{\mathtt{a}}) & = & \{a\} \\ L(r_1 r_2) & = & L(r_1) \cdot L(r_2) \\ & := & \{w_1 w_2 \mid w_1 \in L(r_1), w_2 \in L(r_2)\} \\ L(r_1 \mid r_2) & = & L(r_1) \cup L(r_2) \\ L(r^*) & = & L(r)^* \\ & := & \{w_1 w_2 \ldots w_n \mid n \in \mathbf{N}, w_i \in L(r)\} \\ & = & \{\varepsilon\} \cup L(r) \cup L(r) \cdot L(r) \cup L(r) \cdot L(r) \cdot L(r) \cup \ldots \end{array}$$

A word w belongs to the language described by r iff $w \in L(r)$.

1.3.4 Discussion

We have looked at three different descriptions for regular languages. Now, we assess each method for its usability regarding the specification and implementation of scanners.

A grammar is a low-level means of describing a regular language. A grammar emphasizes the generative aspect of a language definition. While it is easy to generate words in the language from the grammar, it is non-trivial to check if a given word belongs to the language (the word problem). In addition, a grammar is not a concise description of a language. Even simple languages can take many rules to describe. Hence, we conclude that a grammar is neither suited for a high-level specification of a scanner nor for its implementation.

A DFA is also a low-level description of a regular language. Its definition immediately gives rise to a recognizer which is simple to implement efficiently. However, it is not a concise description because even simple languages can require a large set of states in their automaton. In conclusion, while a DFA makes a good implementation it is unsuitable for a high-level specification of a scanner.

A regular expression is a declarative description of a regular language. It can give rise to highly concise descriptions (in particular, if further set-theoretic operations like intersection and difference are included). However, it requires a clever implementation or a translation to a DFA to efficiently recognize words from the language.

Hence, language definitions use 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 automagically 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` | `0O`
⟨octdigit⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`
⟨binprefix⟩ ::= `0b` | `0B`
⟨bindigit⟩ ::= `0` | `1`

Figure 1.1: Some lexical conventions of Caml

1.4 Mapping regular expressions to DFAs

The traditional mapping from regular expressions to DFAs goes through a number of steps. First, a regular expression is mapped to an NFA-ε. Next, the ε-transitions are removed to obtain an NFA. Finally, the "power set construction" is applied to construct an equivalent DFA from the NFA. Moreover, in a typical scanner generator, the resulting DFA is minimized to save space in the implementation.

We follow a slightly different approach, which has been used with variations in implementing regular expression search in text editors. The basic idea is to use a set of regular expressions as the set of states for an automaton. Since we can associate a language to each state q of a finite automaton (the set of words that transform q into a final state), we simply want to label each state with a regular expression for this language. Clearly, the initial state q_0 corresponds to the regular expression that we want to recognize. But what is the regular expression for the state $\delta(q_0, a)$? To address this problem, we define the *derivative* of a regular expression, *i.e.*, a function $D : RE(\Sigma) \times \Sigma \to RE(\Sigma)$ such that

$$w \in L(D(r, a)) \qquad \text{iff} \qquad a\,w \in L(r) \tag{1.1}$$

Thus, if $a\,w \in L(r)$ then $D(r, a)$ recognizes the rest, w, of the input word after reading a. This corresponds to the language recognized by the state $\delta(q_0, a)$.

1.2 Definition
The derivative of a regular expression, $D : RE(\Sigma) \times \Sigma \to RE(\Sigma)$ is defined by induction on the definition of $RE(\Sigma)$. It relies on an auxiliary function $E : RE(\Sigma) \to RE(\Sigma)$ which is specified by

$$L(E(r)) = L(r) \cap \{\varepsilon\} \tag{1.2}$$

$$
\begin{array}{lcl}
D(\underline{\emptyset}, a) & = & \underline{\emptyset} \\
D(\underline{\varepsilon}, a) & = & \underline{\emptyset} \\
D(\underline{\mathtt{a}'}, a) & = & \begin{cases} \underline{\varepsilon} & \text{if } a = a' \\ \underline{\emptyset} & \text{otherwise} \end{cases} \\
D(r_1 r_2, a) & = & D(r_1, a) r_2 \mid E(r_1) D(r_2, a) \\
D(r_1 \mid r_2, a) & = & D(r_1, a) \mid D(r_2, a) \\
D(r^*, a) & = & D(r, a) r^*
\end{array}
$$

$$
\begin{array}{lcl}
E(\underline{\emptyset}) & = & \underline{\emptyset} \\
E(\underline{\varepsilon}) & = & \underline{\varepsilon} \\
E(\underline{\mathtt{a}}) & = & \underline{\emptyset} \\
E(r_1 r_2) & = & E(r_1) E(r_2) \\
E(r_1 \mid r_2) & = & E(r_1) \mid E(r_2) \\
E(r^*) & = & \underline{\varepsilon}
\end{array}
$$

□

With these definitions, we obtain the following representation theorem for a regular language.

1.3 Theorem

$$L(r) = E(r) \cup \bigcup_{a \in \Sigma} a \cdot L(D(r, a))$$

□

Starting from a regular expression r_0 that defines the language we are interested in, it is now easy to define an automaton that recognizes this language.

1.4 Theorem
Let $r_0 \in RE(\Sigma)$. Define the deterministic automaton $M = (Q, \Sigma, \delta, q_0, F)$ as follows:

- *Q is the smallest subset of $RE(\Sigma)$ such that*
  1. *$r_0 \in Q$;*
  2. *if $r \in Q$ and $a \in \Sigma$ then $D(r, a) \in Q$.*
- *$\delta(q, a) = D(q, a)$*
- *$q_0 = r_0$*
- *$F = \{r \in Q \mid \varepsilon \in L(r)\}$.*

Then $L(M) = L(r_0)$.

The problem with this construction is that the set Q may be infinite. To address this problem, we do not use D directly, but insert an additional pass of simplification. Simplfication relies on standard equivalences of regular expressions:

$$
\begin{array}{lcllcl}
r\underline{\emptyset} & = & \underline{\emptyset} r & = & \underline{\emptyset} \\
r\underline{\varepsilon} & = & \underline{\varepsilon} r & = & r \\
r \mid \underline{\emptyset} & = & \underline{\emptyset} \mid r & = & r \\
\underline{\emptyset}^* & = & \underline{\varepsilon}^* & = & \underline{\varepsilon} \\
r^{**} & = & r^*
\end{array}
$$

Using these simplification rules, it is guaranteed that the set of states Q is finite [Brz64] so that the construction actually yields a DFA.

2 Example
For an example, recall part of the regular expression for integer literals from Fig. 1.1.

$$\langle\text{integer-literal}\rangle = (\underline{\varepsilon} \mid \underline{-})\langle\text{digit}\rangle\langle\text{digit}\rangle^*$$

Now

$$\begin{array}{ll} & D(\langle\text{integer-literal}\rangle, -) \\ = & D((\underline{\varepsilon} \mid \underline{-})\langle\text{digit}\rangle\langle\text{digit}\rangle^*, -) \\ = & D(\underline{\varepsilon} \mid \underline{-}, -)\langle\text{digit}\rangle\langle\text{digit}\rangle^* \mid E(\underline{\varepsilon} \mid \underline{-})D(\langle\text{digit}\rangle\langle\text{digit}\rangle^*, -) \\ = & (D(\underline{-}, -) \mid D(\underline{\varepsilon}, -))\langle\text{digit}\rangle\langle\text{digit}\rangle^* \mid (E(\underline{-}) \mid E(\underline{\varepsilon}))D(\langle\text{digit}\rangle\langle\text{digit}\rangle^*, -) \\ = & (\underline{\varepsilon} \mid \underline{\emptyset})\langle\text{digit}\rangle\langle\text{digit}\rangle^* \mid (\underline{\emptyset} \mid \underline{\varepsilon})D(\langle\text{digit}\rangle\langle\text{digit}\rangle^*, -) \\ & \quad \textit{apply simplification} \\ = & \langle\text{digit}\rangle\langle\text{digit}\rangle^* \mid D(\langle\text{digit}\rangle\langle\text{digit}\rangle^*, -) \\ & \quad \textit{last use of } D \textit{ simplifies to } \underline{\emptyset} \textit{ because - is not a } \langle\textit{digit}\rangle \\ = & \langle\text{digit}\rangle\langle\text{digit}\rangle^* \end{array}$$

Hence, after reading a - the automaton still expects a non-empty word of ⟨digit⟩s. It is not a final state because $\varepsilon \notin L(\langle\text{digit}\rangle\langle\text{digit}\rangle^*)$.

In the same way, we can check that

$$\begin{array}{lcl} D(\langle\text{integer-literal}\rangle, +) & = & \underline{\emptyset} \\ D(\langle\text{integer-literal}\rangle, \langle\text{digit}\rangle) & = & \langle\text{digit}\rangle^* \end{array}$$

In the first case, the automaton has reached a *sink state* $\underline{\emptyset}$ which is not a final state and which cannot be left by any transition. In the second case, the automaton has consumed a digit, it has reached a final state but is also ready to read further digits.

1.5 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 regex` contains ambiguities. Specifically, it is possible to make due with regular expressions which contain no internal `Null` constructors; it is always possible to transform a regular expression which is either `Null` or does not contain it 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
```

The regular expression `repeat_one r` is usually written r^+ and it recognizes the language $L(r)^+ = L(r) \cup L(rr) \cup L(rrr) \cup \ldots$, *i.e.*, a finite concatenation of words from $L(r)$ where at least one word is present. The functions `concat_list` and `alternate_list` iterate the concatenation and alternation operators:

$$\begin{array}{lcl} \texttt{concat\_list}\ [r_1; \ldots; r_n] & = & r_1 \ldots r_n \\ \texttt{alternate\_list}\ [r_1; \ldots; r_n] & = & r_1 \mid \cdots \mid r_n \end{array}$$

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 `regexp`. The function `matches` will do exactly that. Its first few lines are straightforward. (It needs to prefix names from module `Regexp` because it resides in a different module.)

```
let rec matches regexp symbols =
  match symbols with
    [] -> Regexp.accepts_empty regexp
  | symbol::rest ->
```

This code calls a function `accepts_empty : 'a regexp -> bool` that checks if the empty sequence belongs to the language of the regular expression. (The implementation of `accepts_empty` is a simple exercise, cf. function E in Sec. 1.4.)

Now that the empty sequence is covered, non-empty sequences are next. This becomes easy in the presence of an auxiliary function `after_symbol`, which implements the derivative function D from Sec. 1.4:

```
after_symbol : 'a -> 'a regexp -> 'a regexp
```

This function has the following behavior:

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

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

Thus, `after_symbol` "subtracts" x from r.

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

```
let next_regexp = Regexp.after_symbol symbol regexp in
if Regexp.is_null next_regexp
then false
else matches next_regexp rest
```

The function `is_null` from `Regexp` checks for the sink state $\underline{\emptyset}$. Its definition is straightforward:

```
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.

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
```

1.6 A real 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. Further, to resolve the ambiguities a scanner is not quite a DFA, but rather needs additional structure.

1.6.1 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` ist still the type of the alphabet of the language, `'token` ist 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`.

1.6.2 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 sink states[1]:

[1] See the appendix A.1.3 for the definition of `filter`.

```
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
```

A scanner description (a list of `lex_rules`) ist 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 =
  let final (regexp, _) = Regexp.accepts_empty regexp in
  filter final 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
```

1.6.3 Resolution of ambiguities

Descriptions of lexical analysis for realistic programming languages almost always contain ambiguities because it is more convenient to specify overlapping lexeme languages. 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.

The standard way of resolving this conflict is the *rule of the longest match*: The first lexeme of a character sequence is its longest prefix which is a lexeme. To

find the longest prefix, even if the scanner recognizes a lexeme, it must continue examining characters of the input until the current prefix is no longer a prefix of a lexeme. Then the scanner returns the last lexeme recognized. This process may involve returning characters to the input.

If there are still two different ways of tokenizing a single lexeme, then the textually preceding rule from the specification if given preference.

1.6.4 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
```

1.6.5 An example specification

As an example, we show some excerpts from the specification of a Caml scanner. In the scanner specification, we implicitly open the `Regexp` module to avoid clutter.

First, there is a number of definitions for regular expressions. Identifiers are specified as follows:

```
let char_range_regexp c1 c2 =
  let int_range = Listplus.from_to (Char.code c1) (Char.code c2) in
  let re i = symbol (Char.chr i) in
  alternate_list(List.map re int_range)

let digit = char_range_regexp '0' '9'
let letter = alternate (char_range_regexp 'A' 'Z') (char_range_regexp 'a' 'z')
let ident_rest = alternate_list [ letter; digit; symbol '_'; symbol '\'' ]
let ident = concat letter (repeat ident_rest)
```

The function `char_range_regexp` takes two characters, `c1` and `c2`, and constructs a regular expression denoting the set of characters between `c1` and `c2`, inclusive.

```
let integer_literal =
  concat (alternate epsilon (symbol '-'))
         (repeat_one digit)
```

An ⟨integer-literal⟩ is an optional sign followed by at least one ⟨digit⟩.

```
let whitespace =
  repeat_one
    (alternate_list
      (List.map symbol [ ' '; '\t'; '\n'; '\r'; '\012'] ))
```

A whitespace lexeme consists of a non-empty sequence of blanks, tabulators, newline, carriage return, and form feed characters.

Next, we define a datatype for the tokens of the Caml language.

```
type caml_token =
    TIdent
  | TInt
  | ...
```

The scanner specification itself is a list of pairs of a regular expression and an action function, as explained above. Typically, we define a scanner as a recursive function, so that it can call itself recursivly to consume further input. This happens in the action for whitespace.

```
let rec token input =
  Lex.scan_one [
    (whitespace,
     function (_, rest) ->
       token rest);
    (ident,
     function (lexeme, rest) ->
       (TIdent,
        Camlsyn.Ident(Ident.from_string (list_to_string lexeme)),
        rest));
    (integer_literal,
     function (lexeme, rest) ->
       (TInt,
        Camlsyn.Int(int_of_string (list_to_string lexeme)),
        rest));
    ...]
    input
```

1.7 Using Ocamllex

The Ocaml language distribution contains a scanner generator suitable for use with Ocaml. At this point, no new concepts are required to understand it, it is only necessary to learn its syntax.

⟨lexer⟩ ::= ⟨code-snippet⟩ ⟨body⟩ ⟨code-snippet⟩
⟨code-snippet⟩ ::= `{` ⟨ocaml-code⟩ `}`
⟨body⟩ ::= ⟨definition⟩* ⟨rules⟩
⟨definition⟩ ::= `let` ⟨ident⟩ `=` ⟨regexp⟩
⟨rules⟩ ::= `rule` ⟨one-rule⟩ (`and` ⟨one-rule⟩)*
⟨one-rule⟩ ::= ⟨ident⟩ `=` `parse` ⟨match⟩ (`|` ⟨match⟩)*
⟨match⟩ ::= ⟨regexp⟩ `{` ⟨ocaml-expression⟩ `}`

Ocamllex translates a scanner specification (described by the non-terminal ⟨lexer⟩) into a corresponding Caml module. The two ⟨code-snippet⟩s are copied literally to the beginning and end of this module, respectively. They can contain arbitrary code, which is not checked during scanner generation.

The ⟨body⟩ of the specification consists of a list of ⟨definition⟩s for regular languages and some ⟨rules⟩. Each definition introduces a name for a regular expression. Definitions cannot be recursive.

Each rule in ⟨rules⟩ specifies a separate scanner function. Scanner functions may call each others recursively. The body of a scanner function is a list of pairs of a regular expression ⟨regexp⟩ and an Ocaml expression. Each regular expression specifies a lexeme language and the associated expression defines the action taken upon finding the lexeme.

Regular expressions are entered using the syntax defined in Sec. 1.3, with a few exceptions and extensions.

- Ocaml character and string constants serve as constant regular expressions.
- The regular expression `_` stands for any single character.
- Character classes are a built-in notion. They are defined by `[` ⟨char-set⟩ `]`, where

⟨char-set⟩ ::= ⟨char-literal⟩
| ⟨char-literal⟩ - ⟨char-literal⟩
| ⟨char-set⟩ ⟨char-set⟩

where ⟨char-literal⟩ stands for a character constant in Ocaml syntax, the second alternative denotes a character range (cf. function `char_range_regexp`), and the thirc alternative denotes the union of two ⟨char-set⟩s.

- Character classes may be complemented by writing `[^` ⟨char-set⟩ `]`.
- An identifier may refer to a preceding definition `let` ⟨ident⟩ `=` ⟨regexp⟩.

Each scanning rule ⟨entry-point⟩ `= parse` ... is translated to a function

`let` ⟨entry-point⟩ `lexbuf = ...`

The parameter `lexbuf` has type `Lexing.lexbuf` (from the standard module `Lexing`) and is the scanners means to access the current input. The action code ⟨ocaml-expression⟩ may refer to `lexbuf` to construct the token or to perform other tasks. The following functions from module `Lexing` are handy in this respect:

- `Lexing.lexeme : Lexing.lexbuf -> string`
  returns the currently matching lexeme
- `Lexing.lexeme_char : Lexing.lexbuf -> int -> char`
  returns the indexed character from the lexeme
- `Lexing.lexeme_start : Lexing.lexbuf -> int`
  returns the absolute position of the start of the current lexeme in the input
- `Lexing.lexeme_end : Lexing.lexbuf -> int`
  returns the absolute position of the end of the current lexeme in the input
- ⟨entry-point⟩ `lexbuf`
  may be called to continue scanning at ⟨entry-point⟩ (which may be different from the current scanner) immediately after the current lexeme.

As an example, we rewrite the fragment of the Caml scanner from Sec. 1.6.5 in Ocamllex syntax.

```
{

type caml_token =
    TIdent
  | TInt
  | ...

}

let digit = ['0' - '9']
let letter = ['A' - 'Z' 'a' - 'z']
let ident_rest = letter | digit | '_' | '\''
let ident = letter ident_rest

let integer_literal = ( | '-') digit digit*
```

```
let whitespace = [ ' ' '\t' '\n' '\r' '\012']+

rule token = parse
    whitespace { token lexbuf }
  | ident { Camlsyn.Ident(Ident.from_string (Lexing.lexeme lexbuf)) }
  | integer_literal { Camlsyn.Int(int_of_string (Lexing.lexeme lexbuf)) }
  | ...
```

The action for `whitespace` does not return a value, instead it calls the scanner `token` recursively. The effect is that whitespace is silently skipped.

1.8 Pragmatic issues

1.8.1 Recognizing keywords

The naive approach at recognizing keywords is to include them as constant regular expressions in the scanner specification. Unfortunately, this can give rise to automata with a huge number of states in the traditional approach and it also leads to inefficiencies in the library-based approach that we are propagating. Hence, keyword recognition is often handled separately from scanning in the following manner.

1. Build a hash table from the keywords before starting the scanner.
2. Specify the scanner so that it recognizes all keywords as identifiers.
3. On recognizing an identifier lexeme, the scanner first checks the hash table. If the lexeme is present it is classified as a keyword. Otherwise, the scanner reports an identifier.

The hash table is constructed only once and its lookup should be performed as quickly as possible. Hence, it is appropriate to spend some effort into its construction. Since all entries of the hash table are a-priori known, it is possible to search for a perfect hash function that avoids collisions. This way, a lookup in the hash table can be guaranteed to run in constant time.

1.8.2 Representing identifiers

Strings are not a good representation for identifiers. In particular, later phases of compilation build so-called symbol tables that map an identifier to some information about it. Since identifier lookups occur very frequently, it is vital that these mappings are implemented efficiently. Each of these lookup operations involves comparison and/or computation of a hash key. Strings perform poor with both types of operation:

- A string comparison takes time linear in the length of the string.
- Computing a (meaningful) hash key for a string is not straightforward, since several characters must be extracted from the string.

Hence, a scanner maps the strings arising as identifier lexemes to *symbols*, first. The usual implementation of this mapping is using an open hashing algorithm. Each new entry in the table is assigned a unique identifier (a number).

These considerations give rise to the following module type `SYMBOL`.

```
type symbol
val symbol : string -> symbol
val name   : symbol -> string
```

A typical implementation for the type `symbol` is a pair of the string representation and a (unique) number.

Chapter 2

Syntactic Analysis

Lexical analysis partitions the program into a sequence of token/attribute pairs. This sequence, however, has no structure as of yet. Therefore, it is necessary to perform further syntactic analysis to reveal more structure. Consequently, language definition manuals describe syntactic structure in terms of higher-level constructs—expressions, statements, declarations etc. Almost without exceptions, the manuals use context-free grammar for this purpose.

⟨type-exp⟩ ::= `'` ⟨ident⟩
| `(` ⟨type-exp⟩ `)`
| ⟨type-exp⟩ `->` ⟨type-exp⟩
| ⟨type-exp⟩ ⟨product-type⟩$^+$
| ⟨type-const⟩
| ⟨type-exp⟩ ⟨type-const⟩
| `(` ⟨type-exp⟩ ⟨type-param⟩$^+$ `)` ⟨type-const⟩
| ⟨type-exp⟩ `as '` ⟨ident⟩
⟨product-type⟩ ::= `*` ⟨type-exp⟩
⟨type-param⟩ ::= `,` ⟨type-exp⟩

Figure 2.1: Partial grammar for Caml type expressions

Figure 2.1 shows a context-free grammar for Caml type expressions. The language defined by the grammar is obviously not regular: it contains recursion. Moreover, tokens from the regular portion of the language syntax show up here effectively as terminals. For simplicity's sake, the grammar employs the lexemes themselves for tokens representing only a single lexeme. Examples are `as`, and `->`.

Syntactic analysis has the following goals:

- Prove the syntactic correctness of a program with respect to a context-free grammar from the language definition. If that is not possible, locate errors and either report or repair them.

- Make the syntax tree of the program accessible to further stages of the compiler.

This last item is quite vague as of yet; clarification will follow later.

2.1 Context-Free Grammars

We need some notation for context-free grammars:

2.1 Definition (Context-Free Grammar)
A context-free grammar *is a tuple* $G = (N, T, P, S)$. *N is the set of* nonterminals, *T the set of* terminals, *$S \in N$ the* start symbol, *$V = T \cup N$ the set of* grammar symbols. *P is the set of* productions*; productions have the form $A \to \alpha$ for a nonterminal A and a sequence α of grammar symbols.*

ϵ is the empty sequence; $|\xi|$ is the length of sequence ξ. Furthermore, α^k denotes a sequence with k copies of α, and $\xi_{|k}$ is the sequence consisting of the first k terminals in ξ.

□

Some letters denote elements of certain sets by default:

$$\begin{aligned} A, B, C, E &\in N \\ \xi, \rho, \tau &\in T^* \\ x, y, z &\in T \\ \alpha, \beta, \gamma, \delta, \nu, \mu &\in V^* \\ X, Y, Z &\in V \end{aligned}$$

All grammar rules in the text are implicitly elements of P.

2.2 Definition (Derives Relation)
G induces the derives relation $\Rightarrow$ *on V^* with*

$$\alpha \Rightarrow \beta \;:\Leftrightarrow\; \alpha = \delta A \gamma \;\wedge\; \beta = \delta \mu \gamma \;\wedge\; A \to \mu$$

and $\overset{*}{\Rightarrow}$ denotes the reflexive and transitive closure of $\Rightarrow$. A derivation *from α_0 to α_n is a sequence $\alpha_0, \alpha_1, \ldots, \alpha_n$ where $\alpha_{i-1} \Rightarrow \alpha_i$ for $1 \leq i \leq n$. A* sentential form *is a sequence appearing in a derivation beginning with the start symbol.*

□

Context-free grammars are easily represented in Caml. Compiler construction actually requires an extended variant of context-free grammars (*attributed* context-free grammars) to be defined later. Suffice it here to say that in an attribute grammar, each production has an additional component, the *attribution*. Both together form a *rule*.

The following type declarations are part of a new structure named `Grammar`.

```
type ('n, 't) symbol =
    NT of 'n
  | T of 't

type ('n, 't) production = P of 'n * (('n, 't) symbol list)

type ('n, 't, 'attrib) rule =
    { production : ('n, 't) production;
      attribution : 'attrib attribution
    }

type ('n, 't, 'attrib) grammar =
    { nonterminals : 'n list;
      terminals : 't list;
```

```
    rules : ('n, 't, 'attrib) rule list;
    start : 'n
  }
```

A few interface functions provide external access to grammars:

```
let nonterminals g = g.nonterminals
let terminals g = g.terminals
let rules g = g.rules
let start g = g.start

let rule_production r = r.production
let rule_lhs r =
  match r.production with P(lhs, _) -> lhs
let rule_rhs r =
  match r.production with P(_, rhs) -> rhs
let rule_attribution r = r.attribution

let rules_with_lhs g n =
  filter (function rule -> rule_lhs rule = n) g.rules
```

2.2 Recursive-Descent Parsing

The introduction of parsing requires sufficient formal background that it is easier to also introduce the relevant algorithm in mathematical notation first. Programs that implement them will follow naturally from the specifications.

2.2.1 Formal Derivation

A simplified notion of a parser is a function which accept s a sequence of terminals if it belongs to the language defined by a grammar, and rejects it otherwise. Consider for each grammar symbol X a function $[X] : T^* \to \mathcal{P}(T^*)$ which fulfills the following equation:

$$[X](x_1 \dots x_n) = \{x_{k+1} \dots x_n \mid X \stackrel{*}{\Rightarrow} x_1 \dots x_k\}$$

Hence, a sequence of terminals ξ is in the language defined by the grammar iff $\epsilon \in [S](\xi)$. Consequently, the above equation specifies an *recognizer* for the grammar. It is, however, not yet suitable for implementation. It leads to one by substituting first terminals, then nonterminals for X. For terminals, the solution is trivial:

$$[x](x_1 \dots x_n) = \begin{cases} \{x_2 \dots x_n\} & \text{if } x_1 = x \\ \varnothing & \text{otherwise} \end{cases}$$

For nonterminals, it is first necessary to extend $[\_]$ to sequences of grammar symbols in a straightforward fashion:

$$[\epsilon](\xi) = \{\xi\}$$
$$[X\alpha](\xi) = \bigcup\{[\alpha](\rho) \mid \rho \in [X](\xi)\}$$

Thus fortified, the definition for nonterminals follows:

$$[A](\xi) = \bigcup_{A \to \alpha} [\alpha](\xi)$$

This last part of the definition recursively descends into the right-hand sides of the grammar rules of a nonterminal. Therefore, this kind of parser is called a *recursive-descent parser*.

It would be a straightforward to translate the above definition into Caml. The result would have two significant problems, however, which would render it unusable:

1. First, the definition really specifies a *non-deterministic* parser: $[A](\xi)$ dives down into the right-hand sides of all grammar productions of A. This actually would cause the implementation to be potentially exponential in the length of the input string!

2. Consider again the fragment of the Caml grammar shown in Fig. 2.1 and substitute into the definition:

$$\begin{aligned}[\langle\text{type-exp}\rangle](\xi) &= \bigcup\{\ldots, [\langle\text{type-exp}\rangle \texttt{ -> } \langle\text{type-exp}\rangle](\xi), \ldots\} \\ &= \bigcup\{\ldots, \bigcup\{[\texttt{-> } \langle\text{type-exp}\rangle](\rho) \mid \rho \in [\langle\text{type-exp}\rangle](\xi)\}, \ldots\}\end{aligned}$$

   Evidently, the definition leads to infinite recursion for grammars which contain so-called *left-recursive* rules—rules which have their left-hand-sides also as the first symbol of their right-hand sides.

The second problem is hard to fix: Recursive-descent parsers do not work for grammars with left-recursive productions. Fortunately, most real programming languages have grammars which can be transformed into an equivalent form without left recursion.

Fortunately, it is possible to fix the exponential blow-up problem. The idea is to change the definition of $[A]$ so that does not union over several right-hand sides but instead picks just one of them immediately. This leads immediately to a linear algorithm. The question is how to settle on a right-hand side—the trick here is to *look ahead* a few terminals in the input without actually parsing and making the decision based this lookahead information associated with the nonterminals of the grammar.

Two functions compute the lookahead information—first_k and follow_k.

2.3 Definition (first_k, follow_k)
For an integer k, first_k and follow_k are defined as follows:

$$\begin{array}{rcl} \text{first}_k & : & V^* \to T^k \\ \text{first}_k(\alpha) & := & \left\{\xi_{|k} \mid \alpha \stackrel{*}{\Rightarrow} \xi\right\} \\ \text{follow}_k & : & N \to T^k \\ \text{follow}_k(A) & := & \{\xi \mid S \stackrel{*}{\Rightarrow} \beta A \gamma \ \wedge \ \xi \in \text{first}_k(\gamma)\} \end{array}$$

□

The first_k function computes, for a sequence of grammar symbols α, all k-sequences of terminals which might result from a derivation starting with α. Then, $\text{follow}_k(A)$ is the set of all terminal sequences of length k that may follow A in a sentential form.

2.4 Lemma
Let G be a context-free grammar without unreachable productions. For a nonterminal A, $\text{follow}_k(A)$ is the smallest solution with $\epsilon \in \text{follow}_k(S)$ of the following fixpoint equation:

$$\text{follow}_k(A) = \text{follow}_k(A) \cup \bigcup_{B \in N} \text{first}_k(\beta\, \text{follow}_k(B)) \mid B \to \alpha A \beta\}$$

□

For any integer k, it is now possible to modify the definition of A in the following way:

$$[A](\xi) = \bigcup_{\substack{A \to \alpha \\ \xi_{|k} \in (\mathrm{first}_k(\alpha)\,\mathrm{follow}_k(A))_{|k}}} [\alpha](\xi)$$

Of course, the above union is ideally over a singleton set. A grammar for which this is always the case for a given k is a so-called *LL(k) grammar*:

2.5 Definition
The LL(k) lookahead *of a production $A \to \alpha$ is computed as follows:*

$$\mathrm{LLLA}_k(A \to \alpha) := (\mathrm{first}_k(\alpha)\,\mathrm{follow}_k(A))_{|k}$$

A context-free grammar G is LL(k) *if, for productions $A \to \alpha$ and $A \to \beta$ with $\alpha \neq \beta$,*

$$\mathrm{LLLA}_k(A \to \alpha) \cap \mathrm{LLLA}_k(A \to \beta) = \varnothing.$$

□

The first "L" is for "parseable from left to right," the second "L" for "choose a production immediately when encountering its left-hand side." (Other textbooks say that the second "L" is really because the parser generates a left derivation—this is a term we can do without here.)

2.2.2 Implementing Recursive-Descent Parsing

With the formal specification at hand, a structure `Ll` for recursive-descent parsing is straightforward to define. Two adaptations are required: The implementation checks for the LL(k) property during parsing and signals conflicts. Also, the $[\_]$ function is split up in the implementation—one version `accept_symbol` for single grammar symbols, one for lists of them called `accept_list`.

```
let accept g k l =
  let first_g_k = Grammar.first g k in
  let follow_g_k = Grammar.follow g k in
  let lookahead rule =
    Grammar.pair_map
      (Grammar.append_truncate k)
      (first_g_k (Grammar.rule_rhs rule))
      (follow_g_k (Grammar.rule_lhs rule))
  in
  let rec accept_symbol symbol l =
    match symbol with
      Grammar.T(t) ->
        (match l with
          [] -> None
        | t'::rest -> if t = t' then Some rest else None)
    | Grammar.NT(n) ->
        let prefix = Grammar.truncate k l in
        let rules =
          filter
            (function rule ->
              List.mem prefix (lookahead rule))
            (Grammar.rules_with_lhs g n)
        in
```

```
        (match rules with
          _::_::_ ->
            print_string "grammar is not LL(";
            print_int k;
            print_string ")\n");
        match rules with
          [] -> None
        | rule::_ -> accept_list (Grammar.rule_rhs rule) l
  and accept_list symbol_list l =
    match symbol_list with
      [] -> Some l
    | symbol::rest ->
        match accept_symbol symbol l with
          None -> None
        | Some l -> accept_list rest l
  in
  accept_symbol (Grammar.NT (Grammar.start g)) l
```

Besides the functions `first` and `follow`, `accept` also uses a few other utilities which go in the `Grammar` structure. The `truncate` functions computes, for an integer k, the k-Prefix of a list:

```
let rec truncate k l =
  match l with
    [] -> []
  | x::xs ->
      if k = 0
      then []
      else x::(truncate (k - 1) xs)
```

The `append_truncate` function is merely a slightly tuned composition of `truncate` and `@`:

```
let rec append_truncate k l1 l2 =
  if k = 0
  then []
  else
    match l1 with
      [] -> truncate k l2
    | x::xs -> x :: (append_truncate (k - 1) xs l2)
```

Lastly, `pair_map` computes all possible pairings of two lists under a binary function. Its type is

```
pair_map : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

and its implementation as follows:

```
let pair_map f l1 l2 =
  let rec loop_1 l1 =
    match l1 with
      [] -> []
    | x::xs ->
        let rec loop_2 l2 =
          match l2 with
            [] -> loop_1 xs
          | y::ys -> (f x y) :: loop_2 ys
        in loop_2 l2
  in loop_1 l1
```

2.3 Recursive-Ascent Parsing

Recursive-descent parsing is simple to implement, but requires, in order to be effective, an LL(k) grammar. Whereas most real programming languages have LL(k) grammars, these are rarely the ones given in a language definition. Usually, substantial changes are required, and the result is rarely as straightforward as the original. (Even more problems arise in the context of attribute grammars—but more about that later.)

Consequently, it is desirable to use a parsing technique which can deal with a larger class of grammars directly—the *recursive-ascent* technique. (This technique is also sometimes known as *bottom-up* or *LR* parsing.) Recursive ascent usually works directly for grammars that occur in programming language definitions. However, it is harder to understand and implement than recursive-descent parsing, and naive implementations lead to slower parsers. Still, it is the most popular technique for automatically generating parsers, probably largely due to the Unix utility `yacc` which generates such parsers.

Again, a formal notation is more suitable for catching the essence of this technique. An implementation follows directly from it. The presentation here follows that in [ST95] and [ST00].

2.3.1 Preliminaries

A deterministic recursive-descent parser always has to know exactly where it is in a grammar. As soon as it encounters a nonterminal, it has to decide on one single production to use for continuing the parsing process. The idea behind recursive-ascent is this: Instead of keeping just *one* position within the grammar as a state, recursive-ascent parsers keep a *set* of such positions around. They only narrow this set down to a single choice once they reach the right-hand side of a grammar production. Because recursive-ascent parsers delay the decision on a grammar production longer than recursive-descent parsers, they are inherently more powerful.

First, recursive-ascent parsers impose a trivial restriction on their input grammars:

2.6 Definition (Start-separated)
A start-separated *context-free grammar $G = (N, T, P, S)$ has just one production with left-hand side S of the form $S \to A$.*

□

From here on, all grammars are start-separated.

A recursive-ascent parser keeps track of its position within the grammar productions with the help of so-called *LR states*. Note that the following definitions take a lookahead size k into account from the very beginning; it has the same meaning here as for recursive-descent parsers.

2.7 Definition (LR(k) item, LR(k) state)
An LR(k) item *(or just* item*) is a triple consisting of a production, a position within its right-hand side, and a terminal string of length k—the* lookahead*. An item is written as $A \to \alpha \cdot \beta\ (\rho)$ where the dot indicates the position, and ρ is the lookahead. If the lookahead is not used (or $k = 0$), it is omitted. A* kernel item *has the form $A \to \alpha \cdot \beta\ (\rho)$ with $|\alpha| > 0$. A* predict item *has the form $A \to \cdot\alpha\ (\rho)$.*

An LR(k) state *(or just* state*) is a non-empty set of LR(k) items.*

□

Whereas the [_] function in recursive-descent parsing operated on a single grammar symbol (or a sequence of them), the equivalent function in recursive-ascent parsing

takes a whole LR(k) state. The initial state of a recursive-ascent parser is q_0 with $q_0 = \{S \to \cdot A\}$.

To investigate the operation of recursive-ascent parsers, a few auxiliary definitions are in order.

2.8 Definition (Predict items, Item transitions, State transitions) *Each state q has an associated set of* predict items*:*

$$\text{predict}(q) := \big\{B \to \cdot\nu\ (\tau) \mid \begin{array}{l} A \to \alpha \cdot \beta\ (\rho) \Downarrow^+ B \to \cdot\nu\ (\tau) \\ \quad \text{for } A \to \alpha \cdot \beta\ (\rho) \in q\big\} \end{array}$$

where $\Downarrow^+$ is the transitive closure of the relation $\Downarrow$ defined by

$$A \to \alpha \cdot B\beta\ (\rho) \Downarrow B \to \cdot\delta\ (\tau) \text{ for all } \tau \in \text{first}_k(\beta\rho).$$

The union of q and predict(q) *is called the* closure *of q. Henceforth,*

$$\overline{q} := q \cup \text{predict}(q)$$

denotes the closure of a state q.

For a state q and a grammar symbol X:

$$\text{goto}(q, X) := \{A \to \alpha X \cdot \beta\ (\rho) \mid A \to \alpha \cdot X\beta\ (\rho) \in \overline{q}\}$$

□

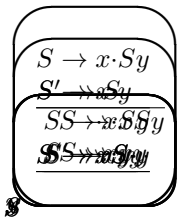


Figure 2.2: LR state diagram for $S \to xSy, S \to xy$

The predict items of a state q are predictions on what derivations the parser may enter next when in state q. The elements of predict(q) are exactly those at the end of leftmost-symbol derivations starting from items in q. All parser states are results of applications of goto. Figure 2.2 shows an example state transition diagram omitting lookahead.

2.3.2 Continuation-Based Recursive Ascent Parsing

It is possible to express LR parsing in a continuation-based form [Spe94].

The parser needs a few new definitions.

2.9 Definition (Next terminals, active symbols)

$$\text{nextterm}(q) := \{x \mid A \to \alpha \cdot x\beta \in \overline{q}\}$$

Each state q has an associated number of active symbols, $\text{nactive}(q)$*:*

$$\text{nactive}(q) := \max\{|\alpha| : A \to \alpha \cdot \beta \in q\}$$

□

When the parser is in state q, then $\text{nactive}(q)$ is the maximal number of states through which the parser may have to return when it reduces by a production in q.

Bunches are a notational convenience for expressing non-deterministic algorithms in a more readable way than a set-based notation [Lee93].

2.10 Definition (Bunch)
A bunch *denotes a non-deterministic choice of values. An atomic bunch denotes just one value. If the a_i are bunches, then $a_1|a_2|\ldots|a_k$ is a bunch consisting of the values of $a_1, \ldots, a_k$. An empty bunch is said to fail and therefore denoted by fail. In other words, $|$ is a non-deterministic choice operator with unit fail. A bunch can be used as a boolean expression. It reads as false if it fails and true in all other cases. Functions distribute over bunches. If a subexpression fails, the surrounding expressions fail as well.*

For bunches P and a, the expression $P \rhd a$ is a guarded expression*: if the guard fails, then the entire expression fails; otherwise the value of $P \rhd a$ is a. It behaves like* ***if*** *P* ***then*** *a* ***else*** *fail.*

□

Figure 2.3 shows the specification of a recursive ascent parser. Here is how it works: The function representing a state contains a *continuation* c_0 which the parser calls whenever it needs to return to that state because it has recognized a production in the derivation. The continuation merely performs a state transition. Now, the function belonging to a state checks if it has recognized a production; this is the case when the dot has reached the end of an item and the lookahead matches. In that case, it needs to go back to the state which introduced the production into the parsing process. The parser finds this state by counting the number of right-hand-side symbols of the production, and calls the corresponding continuation, thereby *ascending* in the call graph. Alternatively, the parser may find that the next input symbol matches a terminal in one of the items in the state; in that case, it calls the current continuation. Figure 2.3 does not quite tell the whole story; namely, it does not specify how the parser ever terminates. In fact, certain *final* states receive special treatment. These are the states in which the input could end legally.

2.11 Definition (Final state)
An LR state is final *if it contains an item of the form $S \to \alpha\cdot$.*

For a final state q_f, the parser definition is augmented with a special rule:

$$[q_f](\epsilon, c_1, \ldots, c_{\text{nactive}(q)}) := \textit{succeed}$$

2.12 Definition (LR(k) grammar)
A grammar for which the parser shown in Fig. 2.3 is deterministic for a given k is called LR(k).

$$
\begin{array}{l}
[q](\xi, c_1, \ldots, c_{\mathrm{nactive}(q)}) := \\
\quad \textbf{letrec} \quad c_0(X, \xi) = \\
\qquad\qquad [\mathrm{goto}(q, X)]\,(\xi, c_0, c_1, \ldots, c_{\mathrm{nactive}(\mathrm{goto}(q,X))-1}) \\
\quad \textbf{in} \qquad A \to \alpha \cdot\ (\rho) \in \overline{q} \wedge \xi_{|k} = \rho \quad \triangleright c_{|\alpha|}(A, \xi) \\
\quad | \qquad \xi = x\xi' \wedge x \in \mathrm{nextterm}(q) \quad \triangleright c_0(x, \xi')
\end{array}
$$

Figure 2.3: Functional LR(k) parser, continuation-based version

2.3.3 Implementing Recursive-Ascent Parsing

Implementing either the direct-style recursive-ascent parser or the continuation-based variant is straightforward, and essentially amounts to transliterating the specification.

The `item` datatype encodes items as a grammar rule together with a position inside the rule along with a position inside the rule and a list of tokens representing the lookahead:

```
type ('n, 't, 'attrib) item =
    Item of (('n, 't, 'attrib) Grammar.rule) * int * 't list
```

Three selectors extract the components of an item:

```
let item_lookahead item = match item with Item(_, _, la) -> la

let item_lhs item = match item with Item(rule, _, _) -> Grammar.rule_lhs rule
let item_rhs item = match item with Item(rule, _, _) -> Grammar.rule_rhs rule
```

The `item_rhs_rest` helper function returns the portion of the right-hand side of an item after the dot:

```
let item_rhs_rest item =
  match item with
    Item(rule, pos, _) -> drop pos (item_rhs item)

let rec drop n l =
  if n = 0
  then l
  else drop (n - 1) (List.tl l)
```

The `item_shift` function shifts the dot of an item by one position to the right:

```
let item_shift item =
  match item with
    Item(rule, pos, la) -> Item(rule, pos+1, la)
```

The `items_merge` merges two lists representing two sets of items:

```
let rec items_merge items_1 items_2 =
  match items_1 with
    [] ->items_2
  | item::items_1 ->
      if List.mem item items_2
      then items_merge items_1 items_2
      else items_merge items_1 (item::items_2)
```

The `predict_equal` function compares two sets of items represented as lists:

```
let predict_equal items_1 items_2 =
  (List.length items_1) = (List.length items_2)
    &&
  let rec loop items_1 =
    match items_1 with
      [] -> true
    | item::items_1 ->
        (List.mem item items_2) && (loop items_1)
  in loop items_1
```

LR states are simply lists of items. The `compute_closure` function computes the closure of a state, given a grammar `g`, lookahead `k`, a function `first_g_k` computing first sets of lists of grammar symbol relative to `g` and `k`:

```
let compute_closure g k first_g_k state =
```

The local function `initial_items` computes, for a nonterminal `n` and lookahead `la_suffix`, a list of items of the form $B \to \delta\nu(\tau)$ where $\tau \in \text{first}_k(\texttt{la\_suffix})$:

```
  let initial_items n la_suffix =
    List.flatten
      (List.map
         (function rule ->
           List.map
             (function la -> Item(rule, 0, la))
             (first_g_k la_suffix))
         (Grammar.rules_with_lhs g n))
  in
```

For a given set of items, `next_predict` computes one step of the $\Downarrow$ relation:

```
  let next_predict item_set =
    let rec loop item_set predict_set =
      match item_set with
        [] -> predict_set
      | item::item_set ->
          match item_rhs_rest item with
            [] -> loop item_set predict_set
          | lhs::rhs_rest ->
              match lhs with
                Grammar.T(t) -> loop item_set predict_set
              | Grammar.NT(n) ->
                  let new_items =
                    initial_items
                      n
                      (rhs_rest @
                         (List.map
                            (function t -> (Grammar.T t))
                            (item_lookahead item)))
                  in
                  loop
                    item_set
                    (items_merge new_items predict_set)
    in loop item_set item_set
```

Finally, the body of `compute_closure` iterators `next_predict` to compute closure:

```
  in let rec loop predict_set =
    let new_predict_set = next_predict predict_set in
    if predict_equal predict_set new_predict_set
    then new_predict_set
    else loop new_predict_set
  in loop state
```

The `goto` function is goto from Definition 2.8:

```
let goto closure symbol =
  List.map
    item_shift
    (filter
       (function item ->
         let rest = item_rhs_rest item in
         (rest != [])
           &&
         (symbol = List.hd rest))
       closure)
```

The `nactive` function is nactive from Definition 2.9:

```
let nactive state =
  let rec loop item_set m =
    match item_set with
      [] -> m
    | Item(_, pos, _)::item_set ->
        loop item_set (max pos m)
  in loop state 0
```

For computing nextterm, it is easiest to start with a function `next_symbols` which computes, for a set of items representing a closure, a list of symbols which appear after the dots:

```
let next_symbols g closure =
  let rec loop item_set symbols =
    match item_set with
      [] -> symbols
    | item::item_set ->
        let rhs_rest = item_rhs_rest item in
        loop
          item_set
          (if (rhs_rest != [] &&
               not (List.mem (List.hd rhs_rest) symbols))
          then (List.hd rhs_rest)::symbols
          else symbols)
  in loop closure []
```

Going from `next_symbols` to `next_terminals` is merely a matter of filtering out the terminals:

```
let is_terminal symbol =
  match symbol with
    Grammar.T(_) -> true
  | Grammar.NT(_) -> false
```

```
let next_terminals g closure =
  List.map
    (function Grammar.T(t) -> t)
    (filter is_terminal (next_symbols g closure))
```

The `accept_items` function filters out those items from a closure that have the dot at the very end:

```
let accept_items closure =
  filter
    (function item -> (item_rhs_rest item) = [])
    closure
```

The `final` function tests if a state could be a final state of the parsing automaton:

```
let final g state =
  let rec loop item_set =
  match item_set with
    [] -> false
  | Item(rule, pos, la)::item_set ->
      (((Grammar.start g) = (Grammar.rule_lhs rule))
         &&
       (pos = List.length (Grammar.rule_rhs rule)))
        or
      (loop item_set)
  in loop state
```

The `start_item` functions constructs the initial state for the parsing automaton:

```
let start_item g =
  Item(List.hd (Grammar.rules_with_lhs g (Grammar.start g)),
       0,
       [])
```

For a given set of items, `select_lookahead_item` selects an item with a lookehead matching an input prefix `l`:

```
let select_lookahead_item k item_set l =
  let prefix = Grammar.truncate k l in
  let matches =
    filter
      (function Item(_, _, la) -> la = prefix)
      item_set
  in
  match matches with
    [] -> None
  | item::_ -> Some item
```

Finally, `accept` is an almost direct transliteration of Figure 2.3:

```
let accept g k l =
  let first_g_k = Grammar.first g k in

  let rec parse state continuations l =
    if (final g state) && l = []
    then true
```

```
    else
      let closure = compute_closure g k first_g_k state in

      let rec c0 symbol l =
        let next_state = goto closure symbol in
        parse
          next_state
          (c0 :: (take ((nactive next_state) - 1) continuations))
          l
      in
      if (l != []) && (List.mem (List.hd l) (next_terminals g closure))
      then match l with t::rest -> c0 (Grammar.T t) rest
      else
        match select_lookahead_item k (accept_items closure) l
        with
          None -> false
        | Some item ->
            (List.nth
               (c0::continuations)
               (List.length (item_rhs item)))
              (Grammar.NT (item_lhs item))
              l
in
parse [start_item g] [] l
```

2.3.4 Simple Lookahead

Pure LR parsers for realistic languages often lead to prohibitively big state automatons [Cha87, ASU86], and thus to impractically big parsers. Fortunately, most realistic formal languages are already amenable to treatment by SLR or LALR parsers which introduce lookahead into essentially LR(0) parsers.

The SLR(k) parser corresponding to an LR(0) parser [DeR71] with states $q_0^{(0)}, \ldots, q_n^{(0)}$ has states closures $q_0, \ldots, q_n$. In contrast to the LR(k) parser, the SLR(k) automaton has the following states:

$$q_i := \{A \to \alpha \cdot \beta \ (\rho) \mid A \to \alpha \cdot \beta \in q_i^{(0)}, \rho \in \text{follow}_k(A)\}$$

Analogously, the predict items are the same as in the LR(0) case, only with added lookahead:

$$\text{predict}(q_i) := \{A \to \alpha \cdot \beta \ (\rho) \mid A \to \alpha \cdot \beta \in \text{predict}^{(0)}(q_i^{(0)}), \rho \in \text{follow}_k(A)\}$$

The state transition goto is also just a variant the LR(0) case here called $\text{goto}^{(0)}$:

$$\text{goto}(q_i, X) := q_j \text{ for } q_j^{(0)} = \text{goto}^{(0)}(q_i^{(0)}, X)$$

It is immediately obvious how to modify a LR(0) parser into an SLR(k) parser—the main parsing function merely has to replace the current state by one decorated by lookahead as described above. The effects of using SLR(k) instead of LR(k) are as expected: generation time and size decrease, often dramatically for realistic grammars.

The LALR method uses a more precise method of computing the lookahead, but also works by decorating an LR(0) parser [DeR69]. Thus, the same methodology as with the SLR case is applicable, merely replacing follow_k with the (more involved) LALR lookahead function. Unfortunately, all efficient methods of computing LALR lookahead sets require access to the entire LR(0) automaton in advance [DP82, PCC85, Ive86, PC87, Ive87b, Ive87a].

2.4 Error Recovery

Realistic applications of parsers require sensible handling of parsing errors. Specifically, the parser should, on encountering a parsing error, issue an error message and resume parsing in some way, repairing the error if possible. The literature abounds with theoretical treatments of recovery techniques applicable to LR parsing [SSS90, Cha87] using a wide variety of methods. Most of the these methods are *phrase-level recovery techniques* that work by transforming an incorrect input into a correct one by deleting terminals from the input and inserting generated ones.

Among the many phrase-level recovery techniques, few have actually been used in production LR parser generators. The widely used Yacc [Joh75] parser generator (as well as its descendant, Bison [DS95]) uses a user-assisted algorithm which is simple, and, for most purposes, quite sufficient.

Yacc provides a special "error terminal" as a means for the user to specify recovery annotations. A typical example is the following grammar for arithmetic expressions:

⟨exp⟩ ::= ⟨term⟩ | `error` | ⟨term⟩ `+` ⟨exp⟩ | ⟨term⟩ `-` ⟨exp⟩
⟨term⟩ ::= ⟨prod⟩ | ⟨prod⟩ `*` ⟨term⟩ | ⟨prod⟩ `/` ⟨term⟩
⟨prod⟩ ::= `number` | `(` ⟨exp⟩ `)` | `(` `error` `)`

Whenever the parser encounters a parsing error, it performs reductions until it reaches a state where it can shift on the error terminal. There, the parser shifts and then skips input terminals until the next input symbol is acceptable to the parser again. In the example, the parser, when encountering an error in a parenthesized expression, will skip until the next closing parenthesis.

To keep the error messages from avalanching, the parser needs to keep track of the number of terminals that have been shifted since the last error; if the last error happened very recently, chances are the new error has actually been effected by the error recovery. In that case, the parser should skip at least one input terminal (to guarantee termination), and refrain from issuing another error message.

This method is fairly crude, but has proven effective for many situations. It also has the advantage over fully automatic methods that it provides the user with the ability to tailor specific error messages to the context of a given error and specify sensible attribute evaluation rules.

In the continuation-based parser, the Yacc method is straightforward to implement. In addition to the usual continuations to perform reductions, we supply an *error continuation* which brings the parser back immediately into the last state to shift on the error terminal. In addition, another parameter keeps track of the number of terminals that the parser still needs to shift until it can resume issuing error messages; in our case that number is three.

2.5 Attributed Grammars

A parser which is just a recognizer is not much good in a compiler—its output conveys nothing about the nature of the parsed input, just if it belongs to the language defined by the underlying grammar or not. In a compiler, a parser needs to communicate the parse tree it has internally generated. Therefore, this section introduces an extended notion of a context-free grammar: the attributed grammar. An attributed grammar associates additional values (the *attribute instances*) with the nodes of a parse tree, and rules that relate them to each other. A parser in a

compiler must compute the values of the attribute instances and return them to the caller.

Attributed grammars carry a considerable amount of notational clutter. Some examples illustrate the central ideas.

Consider again the grammar for constant arithmetic expressions. For technical reasons, only one number has a literal: 42. An attributed grammar can describe how to actually compute the value of such an expression.

$\langle \text{exp} \rangle$ ::= $\langle \text{term} \rangle$	$\langle \text{exp} \rangle.v = \langle \text{term} \rangle.v$
| $\langle \text{term} \rangle$ `+` $\langle \text{exp} \rangle$	$\langle \text{exp} \rangle.v = \langle \text{term} \rangle.v + \langle \text{exp} \rangle.v$
| $\langle \text{term} \rangle$ `-` $\langle \text{exp} \rangle$	$\langle \text{exp} \rangle.v = \langle \text{term} \rangle.v - \langle \text{exp} \rangle.v$
$\langle \text{term} \rangle$ ::= $\langle \text{prod} \rangle$	$\langle \text{term} \rangle.v = \langle \text{prod} \rangle.v$
| $\langle \text{prod} \rangle$ `*` $\langle \text{term} \rangle$	$\langle \text{term} \rangle.v = \langle \text{prod} \rangle.v \cdot \langle \text{term} \rangle.v$
| $\langle \text{prod} \rangle$ `/` $\langle \text{term} \rangle$	$\langle \text{term} \rangle.v = \langle \text{prod} \rangle.v / \langle \text{term} \rangle.v$
$\langle \text{prod} \rangle$ ::= `42`	$\langle \text{prod} \rangle.v = 42$
| `(` $\langle \text{exp} \rangle$ `)`	$\langle \text{prod} \rangle.v = \langle \text{exp} \rangle.v$

$\langle \text{bit} \rangle$ ::= `0`	$\langle \text{bit} \rangle.v = 0$
| `1`	$\langle \text{bit} \rangle.v = 2^{\langle \text{bit} \rangle.s}$
$\langle \text{bits} \rangle$ ::= $\langle \text{bit} \rangle$	$\langle \text{bits} \rangle.v = \langle \text{bit} \rangle.v, \langle \text{bit} \rangle.s = \langle \text{bits} \rangle.s, \langle \text{bits} \rangle.l = 1$
| $\langle \text{bits} \rangle$ $\langle \text{bit} \rangle$	$\langle \text{bits}_1 \rangle.v = \langle \text{bits}_2 \rangle.v + \langle \text{bit} \rangle.v, \langle \text{bit} \rangle.s = \langle \text{bits}_1 \rangle.s,$ $\langle \text{bits}_2 \rangle.s = \langle \text{bits}_1 \rangle.s + 1, \langle \text{bits}_1 \rangle.l = \langle \text{bits}_2 \rangle.l + 1$
$\langle \text{num} \rangle$ ::= $\langle \text{bits} \rangle$	$\langle \text{num} \rangle.v = \langle \text{bits} \rangle.v, \langle \text{bits} \rangle.s = 0$
| $\langle \text{bits} \rangle$ `.` $\langle \text{bits} \rangle$	$\langle \text{num} \rangle.v = \langle \text{bits}_1 \rangle.v + \langle \text{bits}_2 \rangle.v, \langle \text{bits}_1 \rangle.s = 0,$ $\langle \text{bits}_2 \rangle.s = -\langle \text{bits}_2 \rangle.l$

2.5.1 Notation

2.13 Definition (Attributed grammars)
A position *identifies an occurrence of a grammar symbol within a grammar production. The symbol is identified by a $\circ$ directly in front of it. Thus, $\langle \circ A \to \alpha \rangle$ identifies the left-hand side A, whereas $\langle A \to \alpha \circ X \beta \rangle$ identifies the X.*

An attributed grammar *is a tuple $(G, \mathcal{S}, \mathcal{I}, \mathcal{R}, D)$. G is a context-free grammar. $\mathcal{S}$ and $\mathcal{I}$ are families of sets of names indexed by non-terminals: $\mathcal{S} = (\text{Syn}(A))_{A \in N}$, $\mathcal{I} = (\text{Inh}(A))_{A \in N}$. For a non-terminal A, $\text{Syn}(A)$ is the set of* synthesized attributes, *$\text{Inh}(A)$ the set of* inherited attributes *of A. $\text{Att}(A) := \text{Inh}(A) \cup \text{Syn}(A)$ is the set of* attributes *of A. $\text{Syn}(A) \cap \text{Inh}(A) = \varnothing$ is assumed. The notation $A.a$ implies that a is an attribute of A.*

An attribute occurrence *(or just occurrence) is a tuple consisting of a position and an attribute written as $p.a$. An attribute occurrence must either have the form $\langle \circ A \to \alpha \rangle .a$ with $a \in \text{Att}(A)$ or $\langle A \to \gamma \circ B \delta \rangle .a$ with $a \in \text{Att}(B)$.*

Attribute occurrences are naturally associated with a grammar production. The occurrences of a production fall into two classes: the defined *occurrences* $\text{Def}(A \to \alpha)$ *and the* applied *occurrences* $\text{App}(A \to \alpha)$.

$$
\begin{aligned}
\text{Def}(A \to \alpha) :=& \{ \langle \circ A \to \alpha \rangle .a \mid a \in \text{Syn}(A) \} \\
& \cup \{ \langle A \to \gamma \circ B \delta \rangle .a \mid \alpha = \gamma B \delta \ \wedge \ a \in \text{Inh}(B) \} \\
\text{App}(A \to \alpha) :=& \{ \langle \circ A \to \alpha \rangle .a \mid a \in \text{Inh}(A) \} \\
& \cup \{ \langle A \to \gamma \circ B \delta \rangle .a \mid \alpha = \gamma B \delta \ \wedge \ a \in \text{Syn}(B) \}
\end{aligned}
$$

Each defined occurrence d has an associated attribution*—a rule of the form*

$$d = f_d(a_1, \ldots, a_{i_d})$$

where i_d is some natural number associated with d, and all a_j are applied occurrences. f_d must be a function $D^{i_d} \to D$ for the attribute domain *D. $\mathcal{R}$ is the family of all such rules, indexed by the defining attribute occurrences.*

□

Actually, the attribute dependencies of the grammar according to the above definiton are in what other books call *Bochmann normal form* [Boc76]: Inherited attributes depend only on attributes "above" them in the parse tree, synthesized attributes on those below. An attributed grammar assigns meaning to a syntax

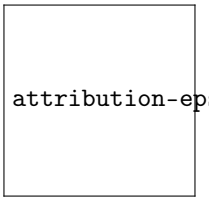


Figure 2.4: A grammar attribution

tree by prescribing how to label its nodes.

2.14 Definition (Attribute labelling)
A attribute labelling *for a syntax trees has the following properties:*

Each nonterminal node n in the syntax trees is labelled with one attribute instance *$n.a$ for each $a \in \mathrm{Att}(A)$. For a node n, let $A \to \alpha$ be the associated production. Furthermore, let, for a position $p = \langle A \to \gamma \circ B\delta \rangle$ with $\gamma B \delta = \alpha$, n^p be the corresponding child node, and, for a position $p = \langle \circ A \to \alpha \rangle$, $n^p = n$. Then, for all $d \in \mathrm{Def}(A \to \alpha)$ with $d = p.a$, the following must hold:*

$$n.a = f_d(n^{p_1}.a'_1, \ldots, n^{p_{i_d}}.a'_{i_d})$$

where $a_j = p_j.a'_j$ for all j.

□

Hence, an attributed grammar G with start production $S \to A$ defines a meaning function $\mathcal{M} : T^* \times D^{|\mathrm{Inh}(A)|} \to D^{|\mathrm{Syn}(S)|}$. For $\xi \in L(G)$, $\mathcal{M}(\xi, v_1, \ldots, v_{|\mathrm{Inh}(A)|})$ seeds the syntax tree with attribute instances for the inherited attributes of A and yields instances of the synthesized attributes of S that result from the labelling of the syntax tree.

There are many techniques for actually producing such a labelling of a syntax tree, some of them quite involved. It is, however, obviously desirable to generate the labelling during parsing. After all, the meaning function is only interested in the attribute instances of the start production; the syntax tree itself is not part of it. Hence, computing the labelling during parsing could avoid having to store the tree. Unfortunately, general attributed grammars may *require* the whole syntax tree to be present for performing attribute evaluation; the attribute rules may, after all, generate circularities.

Therefore, it is necessary to restrict the class of attribute grammars such that they become amenable to "on-the-fly" attribute evaluation. Two particular ways of formulating suitable restrictions on attributed grammars are *L-attributed* and *S-attributed* grammars.

2.15 Definition (L-attributed grammar)
An L-attributed *grammar is an attributed grammar where, for each rule*

$$d = f_d(a_1, \ldots, a_{i_d})$$

with d of the form $\langle A \to \alpha \circ B \beta \rangle .a$ (with $a \in \mathrm{Inh}(B)$), each a_j either has the form $\langle \circ A \to \alpha B \beta \rangle .a$ (with $a \in \mathrm{Inh}(A)$) or $\langle A \to \gamma \circ C \delta B \beta \rangle .a$ (with $\gamma C \delta = \alpha$ and $a \in \mathrm{Syn}(C)$). (Rules with d of the form $\langle \circ A \to \alpha \rangle .a$ have no restrictions.)

In an L-attributed grammar, any attribute occurrence may only depend on ocurrences to its *left* (hence *L*-attributed). Since a recursive-descent parser proceeds from left to right in grammar rules, L-attributed grammars lend themselves to on-the-fly attribute evaluation by recursive-descent parsers.

However, L-attributed grammars present problems for recursive-ascent parsers: As a recursive-ascent parser proceeds forward, it always has to keep several different productions "in mind," each of which may have completely different attribute rules. It therefore would have to either evaluate all of them in parallel—and thus do much superfluous work. Therefore, recursive-ascent parsers usually restrict the class of attribute grammars they accept even further: they simply do not allow inherited attributes.

2.16 Definition (S-attributed grammar)
An S-attributed grammar *is an attributed grammar* $(G, \mathcal{S}, \mathcal{I}, \mathcal{R}, D)$ *with* $\mathrm{Inh}(A) = \varnothing$ *for all* $A \in N$.

For an S-attributed grammar, attribute evaluation always flows upwards in the syntax tree: no circularities may occur, all dependencies point in the same direction. Because of this, it is actually sufficient to have just one synthesized attribute per nonterminal—multiple attributes are easily simulated by using aggregate values such as records. Therefore, the rest of this section assumes $\mathrm{Syn}(A) = \{v\}$ for all $A \in N$. Also, for simplicity of the presentation, $f_{\langle A \to \alpha \rangle .v}$ always has $|\alpha|$ arguments. Terminals simply yield some reserved value $\perp$ as their attribute which may not be used.

$$
\begin{array}{lll}
[q](\xi, c_1, \ldots, c_{\mathrm{nactive}(q)}, v_1, \ldots, v_{\mathrm{nactive}(q)}) := & & \\
\quad \textbf{letrec} & c_0(X, v_0, \xi) = & \\
 & \quad [\mathrm{goto}(q, X)]\,(\xi, c_0, c_1, \ldots, c_{\mathrm{nactive}(\mathrm{goto}(q,X))-1}, v_0, v_1, \ldots, v_{\mathrm{nactive}(\mathrm{goto}(q,X))-1}) & \\
\quad \textbf{in} & A \to \alpha \cdot\ (\rho) \in \overline{q} \wedge \xi_{|k} = \rho & \rhd c_{|\alpha|}(A, f_{\langle A \to \alpha \rangle .v}(v_{|\alpha|}, \ldots, v_1), \xi) \\
\quad | & \xi = x\xi' \wedge x \in \mathrm{nextterm}(q) & \rhd c_0(x, \perp, \xi')
\end{array}
$$

For implementation purposes, it now becomes clear what the `attribution` component of a grammar rule is—it is that function $f_{\langle A \to \alpha \rangle .v}$, and has the following type:

```
type 'attrib attribution = 'attrib list -> 'attrib
```

2.6 Connecting Scanner and Parser

All this talk of parsing has completely ignored the question of lexical analysis so far—it does not figure in the theoretical foundation of parsing. It is obvious, however, that it is necessary to connect the two in some way. The division between lexical and syntactic analysis is little more than a dividing line in the grammar which separates the regular and the non-regular part. This means, however, that the terminals of the non-regular part (the part handled by syntactic analysis), which

are the tokens of the lexical analysis, are in reality nonterminals of the original, complete grammar. As such, they need attributes.

Now it becomes clear that the attributes of the lexical analysis phase really play the same role as the attribute instances in syntactic analysis. It is merely necessary to change the attribute-evaluating parser slightly to take into account that its input it not a sequence of terminals ξ but rather a sequence of terminal("token")/attribute pairs ω. In the following specification, $\pi_1^* : (T \times D)^* \to T^*$ just extracts the terminals from a sequence of terminal/attribute pairs.

$$
\begin{array}{lll}
[q](\omega, c_1, \ldots, c_{\text{nactive}(q)}, v_1, \ldots, v_{\text{nactive}(q)}) := & & \\
\quad \textbf{letrec} & c_0(X, v_0, \omega) = & \\
& \quad [\text{goto}(q, X)]\,(\omega, c_0, c_1, \ldots, c_{\text{nactive}(\text{goto}(q,X))-1}, v_0, v_1, \ldots, v_{\text{nactive}(\text{goto}(q,X))-1}) & \\
\quad \textbf{in} & A \to \alpha \cdot\ (\rho) \in \overline{q} \wedge (\pi_1^*(\omega))_{|k} = \rho & \triangleright c_{|\alpha|}(A, f_{\langle A \to \alpha \rangle . v}(v_{|\alpha|}, \ldots, v_1), \omega) \\
\quad | & \omega = (x, v_0)\omega' \wedge x \in \text{nextterm}(q) & \triangleright c_0(x, v_0, \omega')
\end{array}
$$

2.7 Abstract Syntax

It is possible to fit the complete compilation process in the attribute rules of an attributed grammar. However, when the parser performs attribute evaluation, the attribute evaluation rules are sufficiently constrained to make this quite difficult: Compilation would have to translate constructs in the same order as they are passed—this makes many optimizations impossible. Also, some languages require several passes over a program to resolve names, for example. This is not doable in a natural way using on-the-fly attribute evaluation.

Consequently, it is better to have the attribute rules simply construct a representation of the syntax tree. However, the straightforward representation of a syntax tree is not the most sensible for the purposes of the compiler. Consider the syntax of a subset of Caml called Mini-Caml shown in Fig. 2.5. The syntax contains much "punctuation": parentheses, arrows, infix separators such as `in` or `then` which in a syntax tree simply become "dead leaves:" even if left out, they could be easily reconstructed automatically. Also, the syntax allows for infix expressions which are really just another way of writing binary function application. Beyond the grammar, there is no need to distinguish between prefix and infix application.

Therefore, compilers typically use a more abstract representation of a syntax tree called *abstract syntax*. It still contains all the necessary information to divine the meaning of a program, but omits unnecessary detail. It is possible to reconstruct a program equivalent to the original one from its abstract syntax. As such, the abstract syntax has similar properties to the sequence of token/attribute pairs produced by the scanner. A rule of thumb is that a production in the grammar corresponds to one construct in the abstract syntax. Particulars, however, depend on the particular grammar employed, and on the particular parsing method used, as that may influence the structure of the grammar.

2.17 Definition (Abstract syntax)
Let Σ be the alphabet of the language of a program. Abstract syntax generation *is a function*

$$parse : \Sigma^* \to D$$

where D is a suitable set such that a function

$$unparse : D \to \Sigma^*$$

exists with

$$parse \circ unparse = id_D.$$

⟨exp⟩ ::= ⟨literal⟩
| ⟨ident⟩
| `(` ⟨operator-name⟩ `)`
| `(` ⟨exp⟩ `)`
| ⟨exp⟩ ⟨exp⟩
| ⟨prefix-symbol⟩ ⟨exp⟩
| ⟨exp⟩ ⟨infix-symbol⟩ ⟨exp⟩
| `if` ⟨exp⟩ `then` ⟨exp⟩ `else` ⟨exp⟩
| ⟨exp⟩ `or` ⟨exp⟩
| ⟨exp⟩ `&` ⟨exp⟩
| ⟨exp⟩ `;` ⟨exp⟩
| `function` ⟨ident⟩ `->` ⟨exp⟩
| `raise` ⟨exp⟩
| `try` ⟨exp⟩ `with` ⟨ident⟩ `->` ⟨exp⟩
| `let rec`$^?$ ⟨let-binding⟩ ⟨more-bindings⟩* `in` ⟨exp⟩
| `[` ⟨sequence⟩ `]`

⟨literal⟩ ::= `(` `)` | ⟨integer-literal⟩ | ⟨character-literal⟩ | ⟨string-literal⟩
⟨operator-name⟩ ::= ⟨infix-symbol⟩ | ⟨prefix-symbol⟩
⟨sequence⟩ ::= ⟨empty⟩ | ⟨exp⟩ ⟨sequence-rest⟩*
⟨sequence-rest⟩ ::= `;` ⟨exp⟩
⟨let-binding⟩ ::= ⟨ident⟩$^+$ `=` ⟨exp⟩
⟨more-bindings⟩ ::= `and` ⟨let-binding⟩
⟨definition⟩ ::= `let rec`$^?$ ⟨let-binding⟩ ⟨more-bindings⟩*
⟨program⟩ ::= ⟨definition⟩*

Figure 2.5: Syntax of Mini-Caml

□

In the case of Mini-Caml (which already omits some syntactic frivolities of the full Caml language), the abstract syntax can be even simpler than its concrete syntax. Figure 2.6 shows a Caml type declaration for one such abstract syntax. It has a few peculiarities that warrant explanation:

- It is parameterized over the type of identifiers. This will make it easier to later to deal with automatically generated identifiers, module systems and so forth.
- It distinguished between "normal" identifiers (`Ident`) and built-in ones (`Builtin`) which refer to built-in operations such as primitive arithmetic, storage allocation etc. Even though the parser does not produce `Builtin` nodes, a later analysis can find out which `Ident` nodes actually refer to built-ins.
- It distinguishes between `let` and `let rec`. It will become clear that both need sufficiently different treatment in later phases of the compiler to justify separate abstract syntax constructors.

```
type 'ident syntax =
    Nil
  | Int of int
  | String of string
  | Char of char
  | Ident of 'ident
  | Builtin of 'ident
  | Apply of 'ident syntax * 'ident syntax
  | If of 'ident syntax * 'ident syntax * 'ident syntax
  | Sequence of 'ident syntax * 'ident syntax
  | Function of 'ident * 'ident syntax
  | Raise of 'ident syntax
  | Try of 'ident syntax * 'ident * 'ident syntax
  | Let of 'ident binding * 'ident syntax
  | Letrec of 'ident binding list * 'ident syntax
and 'ident binding = 'ident * 'ident syntax

type 'ident definition = 'ident binding list

type 'ident program = 'ident definition list
```

Figure 2.6: Abstract syntax for Mini-Caml

Chapter 3

The Lambda Calculus

The abstract syntax for Mini-Caml presented in the previous chapter is already fairly compact. However, the real meaning of most of the constructs there is largely unclear. The Caml manual offers an informal description of what each construct does. Unfortunately, this is not sufficient for building a real implementation. In writing a compiler, it is vital to have a precise idea of the meaning of each construct in the language. Therefore, after having attacked Mini-Caml from the front, that is, its appearance to the user, it is now necessary to examine the formal system that forms the basis of Caml, and indeed most other programming languages: the lambda calculus. It will essentially serve as an intermediate language into which the compiler will translate Mini-Caml programs and from which it will generate machine code.

3.1 Syntax and reduction semantics

The lambda calculus is a logical reduction calculus: It consists of a language for terms and a set of reduction rules which describe how to transform terms into other terms.

3.1 Definition (Language of the lambda calculus)
Let $\langle var \rangle$ *be a countable set of* variables. *The following grammar defines the set* $\langle exp \rangle$ *of* lambda terms.

$$\langle \text{exp} \rangle \ ::= \ \langle \text{var} \rangle \ | \ (\ \langle \text{exp} \rangle \ \langle \text{exp} \rangle \ ) \ | \ (\ \lambda \ \langle \text{var} \rangle . \langle \text{exp} \rangle \ )$$

Convention: $x, y, \cdots \in \langle var \rangle$, $e, e_0, e', \cdots \in \langle exp \rangle$.

Terms of the form $(e_0 \ e_1)$ *are* applications, *terms of the form* $\lambda x.e$ *are* abstractions *with* body e.

To save on redundant parentheses, the following conventions apply to the representation of lambda calculus terms:

- *Applications are left-associative.*
- *The body of an abstraction reaches as far to the right as possible.*
- $\lambda xy.e$ *stands for* $\lambda x.\lambda y.e$ *(analogously for more arguments).*

□

Intuitively, the objects of the lambda calculus are functions: An abstraction denotes a function, an application an—application. However, there are different methods for evaluating terms containing functions: first inner terms, then outer, or vice versa, left-to-right, or right-to-left. The lambda calculus, in its original form, has only conversion rules that define a notion of equality between terms. Imposing a direction on the conversion rules turns them into reduction rules and suitable restrictions on where a reduction rule applies will describe such strategies. To properly understand the implications of committing to a particular strategy, it is necessary to first examine the general theory, however.

A description of the meaning of lambda terms requires some auxiliary definitions.

3.2 Definition (Free and bound variables)
The functions $\mathrm{free}, \mathrm{bound} : \langle \mathit{exp} \rangle \to \mathcal{P}(\langle \mathit{var} \rangle)$ *return the set of* free *or* bound *variables of a lambda term, respectively.*

$$\begin{aligned} \mathrm{free}(x) &:= \{x\} \\ \mathrm{free}(e_0\ e_1) &:= \mathrm{free}(e_0) \cup \mathrm{free}(e_1) \\ \mathrm{free}(\lambda x.e) &:= \mathrm{free}(e) \setminus \{x\} \end{aligned}$$

$$\begin{aligned} \mathrm{bound}(x) &:= \varnothing \\ \mathrm{bound}(e_0\ e_1) &:= \mathrm{bound}(e_0) \cup \mathrm{bound}(e_1) \\ \mathrm{bound}(\lambda x.e) &:= \mathrm{bound}(e) \cup \{x\} \end{aligned}$$

Furthermore, $\mathrm{var}(e) := \mathrm{free}(e) \cup \mathrm{bound}(e)$ *is the* set of variables *of* e*. A lambda term* e *is* closed *(*e *is a* combinator*) iff* $\mathrm{free}(e) = \varnothing$*.*

□

3.3 Definition (Substitution)
For $e, f \in E$*,* $e[x \mapsto f]$ *is inductively defined by:*

$$\begin{aligned} x[x \mapsto f] &:= f \\ y[x \mapsto f] &:= y && x \neq y \\ (\lambda x.e)[x \mapsto f] &:= \lambda x.e \\ (\lambda y.e)[x \mapsto f] &:= \lambda y.(e[x \mapsto f]) && x \neq y, y \notin \mathrm{free}(f) \\ (\lambda y.e)[x \mapsto f] &:= \lambda y'.(e[y \mapsto y'][x \to f]) && x \neq y, y \in \mathrm{free}(f), y' \notin \mathrm{free}(e) \cup \mathrm{free}(f) \\ (e_0\ e_1)[x \mapsto f] &:= (e_0[x \mapsto f])(e_1[x \mapsto f]) \end{aligned}$$

□

3.4 Definition (Reduction rules)
There are three different reduction rules for the lambda calculus: α *reduction,* β *reduction, and* η *reduction. Each is a binary relation on lambda terms.*

$$\begin{aligned} \lambda x.e &\to_\alpha \lambda y.e[x \mapsto y] && y \notin \mathrm{free}(e) \\ (\lambda x.e)\ f &\to_\beta e[x \mapsto f] \\ (\lambda x.e\ x) &\to_\eta e && x \notin \mathrm{free}(e) \end{aligned}$$

Each reduction rule is compatibly extended to also work on subterms. That is,

$$\frac{e \to_x e'}{\lambda y.e \to_x \lambda y.e'} \qquad \frac{e_0 \to_x e_0'}{(e_0\ e_1) \to_x (e_0'\ e_1)} \qquad \frac{e_1 \to_x e_1'}{(e_0\ e_1) \to_x (e_0\ e_1')}$$

For $x \in \{\alpha, \beta, \gamma\}$*,* $\overset{*}{\to}_x$ *is the reflexive-transitive closure, and* $\leftrightarrow_x$ *is its symmetric closure, and* $\overset{*}{\leftrightarrow}_x$ *is its reflexive-transitive-symmetric closure.*

□

Every term matching the left side of a reduction rule is a *redex*.

A β-reduction step corresponds closely to the intuitive notion of function application.

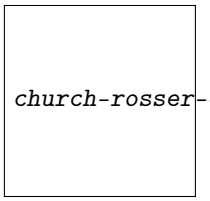
Lambda terms will be considered equivalent if only the names of their bound variables differ (i.e., if they are α-convertible). If variable names matter, $e \equiv e'$ indicates that e and e' are identical, including the names of bound variables.

3.5 Definition (Normal form)
Let e be a lambda term. A lambda term e' is a normal form *of e iff $e \overset{*}{\to}_\beta e'$ and if there is no e'' with $e' \to_\beta e''$.*

Lambda terms with equivalent (equal modulo α reduction) normal forms exhibit the same behavior. The reverse is not always true. Also, some lambda terms do not have a normal form:

$$(\lambda x.x\ x)(\lambda x.x\ x) \to_\beta (\lambda x.x\ x)(\lambda x.x\ x)$$

3.6 Theorem (Church-Rosser)
The β reduction has the Church-Rosser property*:*



In words: For all lambda terms e_1, e_2 with $e_1 \overset{*}{\leftrightarrow}_\beta e_2$, there is a lambda term e' with $e_1 \overset{*}{\to}_\beta e'$ and $e_2 \overset{*}{\to}_\beta e'$.

□

3.7 Corollary (Corollary)
A lambda term e has at most one normal form modulo α reduction.

□

3.2 Programming in the lambda calculus

The lambda calculus may at first seem a fairly silly way to go about describing functions: In the world of the lambda calculus, there is nothing *but* functions, and the scarcity of its language seems to allow for only the most primitive computations (if any). Nevertheless, the lambda calculus has the same computational power as any programming language. (The theoreticians say it is "Turing-equivalent.")

Adding conventional programming language constructs to the lambda calculus is somewhat tedious. As such, these constructions are not readily usable for programming language implementations. However, it is good to have a working knowledge of the necessary mechanisms if only to get some practice dealing with the calculus. In practice, the "pure" lambda calculus gives way to an "applied" lambda calculus which has the necessary built-in data types and primitive operations on them to directly perform useful computations.

What constructs are necessary for useful computations? The lambda calculus at first glance seems to lack the following fundamental ingredients:

- some sort of conditional and booleans,
- numbers, and

- recursion.

These are (almost) the elements of the theory of recursive functions. An applied lambda calculus typically has all of these, but it is possible to model all of them (and more) in the pure one. This yields an informal proof that every recursive function on natural numbers can be encoded in a lambda term. Consequently, every Turing machine can be simulated by a lambda term. To prove Turing-equivalence, as hinted above, it is now sufficient to implement β-reduction on a Turing machine.

3.2.1 Booleans and conditionals

Conditionals in functional languages usually have the form `if` e `then` e_1 `else` e_2: Depending on the (boolean) result of evaluating e, the conditional "selects" either e_1 or e_2. The way to go in the lambda calculus is to give booleans themselves an "active" interpretation that *performs* the selection by itself. Thus, *true* is a lambda term that selects the first of two arguments, and *false* is one that selects the second:

$$\begin{aligned} \mathit{true} &:= \lambda xy.x \\ \mathit{false} &:= \lambda xy.y \end{aligned}$$

Consequently, the conditional degenerates to an identity function:

$$\mathit{if} := \lambda txy.t\ x\ y$$

It is straightforward to verify that *if* actually adheres to the intuition. For a true test, the beta reduction goes like this:

$$\begin{aligned} \mathit{if}\ \mathit{true}\ e_1\ e_2 &= (\lambda txy.t\ x\ y)\ \mathit{true}\ e_1\ e_2 \\ &\to_\beta (\lambda xy.\mathit{true}\ x\ y)\ e_1\ e_2 \\ &\to_\beta^2 \mathit{true}\ e_1\ e_2 \\ &= (\lambda xy.x)\ e_1\ e_2 \\ &\to_\beta (\lambda y.e_1)\ e_2 \\ &\to_\beta e_1 \end{aligned}$$

For *false*, the proof goes analogously.

3.2.2 Numbers

Numbers can be represented in several different ways by lambda terms. One is to use *Church numerals*. The Church numeral $\lceil n \rceil$ of some natural number n is a function which takes two parameters, a function f and some x, and applies f n-times to x. (Hence, $\lceil 0 \rceil$ is the identity.)

$$\lceil n \rceil := \lambda f \lambda x.f^{(n)}(x)$$

where

$$f^{(n)}(e) := \begin{cases} e & \text{if } n = 0 \\ f(f^{(n-1)}(e)) & \text{otherwise} \end{cases}$$

The successor function adds an application:

$$\mathit{succ} := \lambda n.\lambda f \lambda x.n\ f\ (f\ x)$$

The predecessor is somewhat more complicated:

$$\mathit{pred} := \lambda x.\lambda y.\lambda z.x\ (\lambda p.\lambda q.q\ (p\ y))\ ((\lambda x.\lambda y.x)\ z)\ (\lambda x.x)$$

(A proof that it actually does subtract one from a Church numeral is a worthwhile exercise.)

Also, a test for zero is possible:

$$\mathit{zero?} := \lambda n.n\ (\lambda x.\mathit{false})\ \mathit{true}$$

Again, a simple test case serves as an example:

$$\begin{aligned}\mathit{zero?}\ \lceil 0\rceil &= (\lambda n.n\ (\lambda x.\mathit{false})\ \mathit{true})\ \lceil 0\rceil\\ &\to_\beta \lceil 0\rceil\ (\lambda x.\mathit{false})\ \mathit{true}\\ &= (\lambda f.\lambda x.x)\ (\lambda x.\mathit{false})\ \mathit{true}\\ &\to_\beta (\lambda x.x)\ \mathit{true}\\ &\to_\beta \mathit{true}\end{aligned}$$

3.2.3 Recursion

The only thing missing now is recursion. Since a recursive function needs to refer to itself, it needs to receive a name which is passed to it by a magical term called a *fixpoint combinator*. The magic is sufficient to warrant a theorem:

3.8 Theorem (Fixpoint theorem)
Every lambda term has a fixpoint.

That is, for every lambda term f there is a lambda term e with $f\ e \leftrightarrow\!\!\!\!\!\rightarrow^*_\beta e$.

Proof:

Choose $e := Y\ f$ with

$$Y := \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)).$$

Then:

$$\begin{aligned}Y\ F &= (\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))\ F\\ &\to_\beta (\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x))\\ &\to_\beta F\ ((\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x)))\\ &\leftarrow_\beta F\ ((\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ F)\\ &= F\ (Y\ F)\end{aligned}$$

□

A fixpoint combinator suitable for multiple recursion involves no new principles, but is very nasty to formulate.

As an example, consider expressing the recursive definition of the factorial function

$$\mathit{fac}\ n = \mathit{if}\ (\mathit{zero?}\ n)\ \lceil 1\rceil\ \mathit{times}\ n\ (\mathit{fac}\ (\mathit{pred}\ n))$$

where *times* and *pred* are multiplication and predecessor functions. An equivalent non-recursive definition can be found using the fixpoint combinator.

$$\mathit{fac}' = Y\ (\lambda f\ n.\mathit{if}\ (\mathit{zero?}\ n)\ \lceil 1\rceil\ \mathit{times}\ n\ (f\ (\mathit{pred}\ n)))$$

It is straightforward to show that, for all $n \in \mathbb{N}$, $\mathit{fac}\ \lceil n\rceil \leftrightarrow\!\!\!\!\!\rightarrow^*_\beta \mathit{fac}'\ \lceil n\rceil$.

3.2.4 Pairs

Other data structures are readily implementable, too. For example, a pair can be encoded as a function that takes a projection function and applies it to the components of the pair. Hence, the selectors take a pair and apply it to the appropriate projection function.

$$\begin{aligned} \mathit{pair} &:= \lambda xyt.t\ x\ y \\ \mathit{fst} &:= \lambda p.p\ \lambda xy.x \\ \mathit{snd} &:= \lambda p.p\ \lambda xy.y \end{aligned}$$

3.3 Evaluation strategies

Since vanilla β reduction applies to arbitrary subterms, having normal forms is of limited value: It is not clear how to compute them because success is highly dependent on the order in which subterms are subject to reduction. In the practice of programming, full normal forms are rarely important. Instead, it is usually sufficient to evaluate lambda terms to the point where they are simple values or abstractions; it is not necessary to evaluate anything "'inside the lambda."' This leads to the notion of *weak head-normal forms*:

3.9 Definition (Weak head-normal form)
A lambda term which is an abstraction is called a value *or a* weak head-normal form. *All other lambda terms are called* expression juxtapositions.

□

Next, it is desirable to formulate deterministic strategies that prescribe how to evaluate a λ term to its weak head-normal form, so-called *evaluation strategies*. A succinct formalism for describing such strategies is the concept of an evaluation context:

3.10 Definition (Evaluation contexts)
An evaluation context C *is a mapping* $\langle exp \rangle \to \langle exp \rangle$ *denoted by a lambda term with a "hole," i.e. one of its subterms is the literal* $[\,]$. *The mapping works by replacing the hole by its argument. The notation for "plugging" a lambda term* e *into such a hole is* $C[e]$. *The following defines the plugging operation:*

$$\begin{array}{lcll} [\,][e] & = & e & \text{where } e \in \langle exp \rangle \\ (\lambda x.C)[e] & = & \lambda x.(C[e]) & C \text{ is an evaluation context} \\ (C\ e')[e] & = & ((C[e])\ e') & C \text{ is an evaluation context} \\ (e'\ C)[e] & = & (e'\ (C[e])) & C \text{ is an evaluation context} \end{array}$$

Unlike substitution, the plugging operation does not rename bound variables.

□

Furthermore, for each notion of evaluation, a suitable set of lambda terms must be identified as answers or values, which qualify as results of an evaluation. To make sense, values should not evaluate further. In addition, evaluation is only defined for closed lambda terms. Evaluation contexts must be defined such that every lambda term e falls in one of the following categories.

- e is a value;
- e can be uniquely written as $E[r]$ where E is an evaluation context and r is a redex;

- e is stuck (it cannot be reduced further).

The last case does not apply to the pure lambda calculus. In the applied calculus, it corresponds to a type mismatch.

Hence, we start with defining a notion of value.

3.11 Definition (Weak head-normal form)
A lambda term is in weak head-normal form *iff it has the form*

$$v ::= \lambda x.e$$

□

Weak head-normal forms are suitable as *values* for all common evaluation strategies. All other terms (of the form $e_0\ e_1$) are *non-values*.

The two important strategies for computing weak head-normal forms are called *call-by-name* and *call-by-value*:

3.12 Definition (Call-by-name lambda calculus)
Call-by-name evaluation contexts are defined by

$$E_n ::= [\,] \mid E_n\ e.$$

The call-by-name one-step evaluation relation $\to_{\beta n}$ *is defined by:*

$$E_n[(\lambda x.e)\ f] \to_{\beta n} E_n[e[x \mapsto f]].$$

The call-by-name evaluation function is a partial function $\mathbf{eval}_n : \langle exp \rangle \to \langle exp \rangle$ *where* $\mathbf{eval}_n(e) = v$ *iff there exists a value* v *such that* $e \to^*_{\beta n} v$.

□

Unfortunately, the call-by-name lambda calculus is of limited value for programming language implementation—it may evaluate a subexpression multiple times. Real-world programming languages either use a refinement of the call-by-name strategy to avoid multiple evaluation (lazy evaluation) or use a call-by-value strategy which evaluates arguments to lambda abstractions before β-reducing them.

3.13 Definition (Call-by-value lambda calculus)
Call-by-value evaluation contexts are defined by

$$E_v ::= [\,] \mid (E_v\ e) \mid (v\ E_v).$$

The call-by-value one-step evaluation relation $\to_{\beta v}$ *on lambda terms is defined by*

$$E_v[(\lambda x.e)\ v] \to_{\beta v} E_v[e[x \mapsto v]].$$

The call-by-value evaluation function is a partial function $\mathbf{eval}_v : \langle exp \rangle \to \langle exp \rangle$ *where* $\mathbf{eval}_v(e) = v$ *iff there exists a value* v *such that* $e \to^*_{\beta v} v$.

□

In this definition, β-reduction is restricted to a β_v-reduction where the argument position in the redex is already a value.

None of the above strategies is guaranteed to compute a normal form for the original unrestricted notion of β-reduction. Therefore, theoretician consider less restrictive strategies, normal-order reduction and applicative-order reduction, which have different properties.

Normal-order reduction always finds the normal form of a lambda term if it has one. Intuitively, it corresponds to always choosing the leftmost-outermost β-redex. To specify it, the final result of such an evaluation must be described. Hence, answers are now normal forms of lambda terms. The following grammar generates normal forms in the non-terminal $\langle \mathrm{nf} \rangle$. They are ranged over by n.

⟨nf⟩ ::= ⟨nf'⟩ | (λ ⟨var⟩ . ⟨nf⟩)

⟨nf'⟩ ::= ⟨var⟩ | (⟨nf'⟩ ⟨nf⟩)

3.14 Definition (Normal-order reduction)
Leftmost-outermost evaluation contexts are defined by

$$E_o ::= [\,] \mid E_o\ e \mid \lambda x.E_o$$

The normal-order reduction relation $\rightarrow_{\beta o}$ *(also called* leftmost-outermost reduction *or* standard reduction*) is defined by:*

$$E_o[(\lambda x.n)\ f] \rightarrow_{\beta o} E_o[n[x \mapsto f]].$$

□

There is a similar generalization for call-by-value evaluation.

3.15 Definition (Applicative-order reduction)
Leftmost-innermost evaluation contexts are defined by

$$E_i ::= [\,] \mid (E_i\ e) \mid (n\ E_i) \mid \lambda x.E_i.$$

The applicative-order reduction relation $\rightarrow_{\beta i}$ *(*leftmost-innermost reduction*) on lambda terms is defined by*

$$E_i[(\lambda x.n)\ n'] \rightarrow_{\beta i} E_i[n[x \mapsto n']].$$

□

3.4 Recursive applicative program schemes

Neither Church numerals nor the fixpoint combinator are particularly efficient ways of implementing realistic programs. Therefore, real programming languages incorporate the lambda calculus in some applied form which already contains essential primitive data types and their operations as well as recursion. One particular way of formulating an applied lambda calculus is to use *recursive applicative program schemes*. Such a program is essentially a set of equations, the right-hand sides of which are lambda terms.

3.16 Definition (Recursive applicative program schemes)
Let ⟨const⟩ be a countable set of names for constants *with ⟨var⟩ ∩ ⟨const⟩ = ∅. Let ⟨fname⟩ be a countable set of* function names*, disjoint from ⟨var⟩ and ⟨const⟩. An* applied lambda term *is a term generated by the following grammar:*

⟨exp'⟩ ::= ⟨var⟩
| ⟨fname⟩
| ⟨const⟩
| (λ ⟨var⟩ . ⟨exp'⟩)
| (⟨exp'⟩ ⟨exp'⟩)
| *let* ⟨var⟩ = ⟨exp'⟩ *in* ⟨exp'⟩
| *if* ⟨exp'⟩ *then* ⟨exp'⟩ *else* ⟨exp'⟩

The applied lambda terms form a set E'.

A recursive applicative program scheme *is a set of equations together with a lambda term with the following form:*

⟨scheme⟩ ::= ⟨equation⟩* ⟨exp'⟩
⟨equation⟩ ::= ⟨fname⟩ = ⟨exp'⟩

The recursive applicative program schemes form a set ⟨scheme⟩.

□

Evaluation of a program scheme proceeds on the expression at the end of the scheme. The new constructs *let*, *if*, and ⟨fname⟩ have new reduction rules and extend the evaluation contexts.

- *let* is reduced according to

$$let\ x\ =\ e'\ in\ e \equiv (\lambda x.e)\ e'$$

- The above definition for program schemes relies on constants *true* and *false* for booleans. The *if* is reduced in the intuitive way:

$$if\ true\ then\ e_1\ else\ e_2 \to e_1$$
$$if\ false\ then\ e_1\ else\ e_2 \to e_2$$

  Evaluation contexts for call-by-name and call-by-value extend by

$$E_x ::= \cdots \mid if\ E_x\ then\ e_1\ else\ e_2$$

- A function name in a lambda term must correspond to an equation; it reduces to the right-hand side of that equation when the name appears in an evaluation context. Some languages —especially call-by-value ones— insist in binding names only to values. In this case, the ⟨scheme⟩ itself has to take some reduction steps before the body expression can start evaluation.

- For constants, the "user" must define additional reduction rules, so-called δ reductions which depend on their concrete meaning. Usually, each constant $c^{(a)}$ has an arity, a, and its δ reduction is defined by a partial function δ_c which maps an a-tuple of terms in WHNF to a term in WHNF. For instance, if a scheme were to support integers and addition, there would be a constant $\lceil n \rceil$ for each integer n as well as a constant $+^{(2}$ with δ_+ defined by:

$$\delta_+(\lceil m \rceil, \lceil n \rceil) = \lceil m + n \rceil$$

  The δ reduction relation is generically defined by

$$c^{(a)}\ v_1 \ldots v_a \to_\delta v \quad \text{if} \quad ((v_1, \ldots, v_a), v) \in \delta_c$$

  Every non-nullary constant generates new terms in WHNF and new evaluation contexts. They are identical for call-by-name and call-by-value semantics.

$$v \quad ::= \quad \cdots \mid c^{(0)} \mid c^{(1)} \mid c^{(2)} \mid c^{(2)}\ v \mid \ldots c^{(a)} \mid c^{(a)}\ v \mid \ldots c^{(a)}\ \underbrace{v \ldots v}_{a-1} \ldots$$
$$E_x \quad ::= \quad \cdots \mid c^{(a)}\ \underbrace{v \ldots v}_{a'}\ E_x \quad \text{for all} \quad a' < a$$

  In the example of the + operator this amounts to

$$v \quad ::= \quad \cdots \mid + \mid + v$$
$$E_x \quad ::= \quad \cdots \mid + E_x \mid + v\ E_x.$$

3.4.1 Execution errors

In an applied lambda calculus, there are usually terms which cannot be evaluated further although they are not in weak head-normal form. These terms are called "stuck terms". They are regarded as execution errors because they amount to misinterpretation of data. Here are some examples.

$\lceil 5 \rceil\ v$	number used as a function, arity mismatch
$+\ (\lambda x.e)\ v$	operand out of domain
if $(\lambda x.e)$ *then* e_1 *else* e_2	type mismatch
if $\lceil 42 \rceil$ *then* e_1 *else* e_2	type mismatch

Programming languages take one of two stances. Either they expect the compiler to generate code that tests all operands before it executes an operation. Or they impose a typing discipline that rules out some or all programs that may lead to execution errors. The first case is often called *dynamic typing* or *dynamic checking* and it requires that every value is equipped with sufficient type information at runtime. The other case amounts to *static typing* or *static checking* and it imposes on the compiler writer the burden of implementing a type checker to enforce the typing discipline. Depending on the discipline, this task can range from straightforward through demanding to impossible (there are some typing disciplines with undecidable type checking).

3.5 Denotational semantics of the lambda calculus

Reduction rules are a convenient way of giving meaning to lambda terms. Unfortunately, they are not suitable as a basis for an implementation of the lambda calculus. Programming language implementation by reduction is inefficient. However, denotational semantics provide the necessary formalism to give precise meaning to a lambda calculus term (or, rather, a recursive program scheme) while still being suitable for the straightforward derivation of an implementation.

3.5.1 Domains for semantics

The central idea in denotational semantics is to assign a mathematical *object* to a program—rather than a behavior as specified by reduction rules. Formulating a denotational semantics for a programming language as a first step requires specifying the mathematical domains used in the semantics. Much theoretical background exists on such domains, but an expository treatment suffices for our purposes.

!!! TO BE CONTINUED

What is a continuous function?

The first step in forming such domains is to recognize that describing recursion requires computing fixpoints. Computation of fixpoints requires some sort of fixpoint iteration over a mathematical structure with some partial order. The partial order intuitively describes the amount of information in a value; fixpoint iteration must increase this amount upon every iteration to guarantee that it will compute a meaningful value.

For a domain D, the order will usually be written as $\sqsubseteq_D$. As a starting point for fixpoint iteration (and as a prerequisite for the existence of fixpoints) a semantical domain D must contain a least element $\bot_D$ (pronounced "bottom") that stands for a non-terminating computation—an undefined value. Domains are often defined using equations, which can be recursive. Domain theory guarantees these recursive equations have a smallest solution.

Of course, realistic computation requires the assembly of larger domains from smaller ones. Here are a few constructions used in the semantics for program schemes:

Primitive domains

- $\mathbb{Z}_\perp$ is the flat domain of integers consisting of $\perp_{\mathbb{Z}_\perp}, 0, 1, -1, \ldots$.
- $\mathbb{T}_\perp$ is the flat domain of booleans consisting of $\perp_{\mathbb{T}}$, *true*, and *false*. The expression $e \to e_1, e_2$ denotes a conditional: If e denotes *true*, the expression's denotation is that of e_1, in the case of *false*, it is that of e_2, and $\perp$ otherwise.

Lifted domains

- $D_\perp$ denotes a domain D' obtained by adding a new least element $\perp_{D'}$ under a "copy" of D.
- $\mathrm{up}_D : D \to D_\perp$ maps each element of D to the corresponding element of $D_\perp$.
- $\mathrm{down}_D : D_\perp \to D$ maps each element of $D_\perp$ back to the corresponding element of D and $\perp_{D_\perp}$ to $\perp_D$.

Product domains

- $D_1 \times \cdots \times D_n$ is the Cartesian product of the domains. The elements of such a product have the form $(e_1, \ldots, e_n)$. The order of the product is defined by $(d_1, \ldots, d_n) \sqsubseteq (d'_1, \ldots, d'_n)$ iff $d_i \sqsubseteq_{D_i} d'_i$, for all $1 \le i \le n$.
- $D_1 \otimes \cdots \otimes D_n$ denotes the *smash product* obtained from the Cartesian product by identifying all tuples which contain a $\perp$ component with a new smallest element $\perp$.
- $\mathrm{smash}_D : D_1 \times \cdots \times D_n \to D_1 \otimes \cdots \otimes D_n$ maps each element of a Cartesian product into the corresponding smash product.
- $\mathrm{unsmash}_D : D_1 \otimes \cdots \otimes D_n \to D_1 \times \cdots \times D_n$ maps each element of a smash product into the corresponding Cartesian product.

Sum Domains

- $D_1 + \cdots + D_n$ is the separated sum of the domains D_i whose elements are distinguished copies of the elements of the D_i together with a new $\perp_{D_1+\cdots+D_n}$.
- $D_1 \oplus \cdots \oplus D_n$ is the coalesced sum domains obtained from the separated sum by identifying all copies of $\perp_{D_i}$ elements with a new element $\perp$.
- in_i^D is an injection from a summand D_i of $D = D_1 \oplus \cdots \oplus D_n$ into D.
- $[e_1, \ldots, e_n] : D_1 \oplus \cdots \oplus D_n \to D'$ is the case analysis of functions $f_i : D_i \to D'$ denoted by the e_i mapping $\mathrm{in}_i^D(x)$ (with $D = D_1 \oplus \cdots \oplus D_n$) to $f_i(x)$ and $\perp_{D_1 \oplus \cdots \oplus D_n}$ to $\perp_{D'}$.

Function Domains

- $D_1 \to D_2$ is the domain of all continuous functions from D_1 to D_2 under the pointwise ordering: $f \sqsubset_{D_1 \to D_2} g$ iff $f(x) \sqsubseteq_{D_2} g(x)$, for all $x \in D_1$.
- $\lambda x \in D.e$ denotes a continuous function f given by defining $f(x) := e$ where x ranges over D. (The notational clash with the lambda calculus is unfortunate, but pervasive in the literature.)
- $f\ e$ denotes the result of applying f to e.
- lfp_D is the least fixpoint operator for a domain D which maps a function f in $D \to D$ to the least solution of $x = f(x)$.
- $D_1 \circ\!\!\to D_2$ is the restriction of $D_1 \to D_2$ to strict functions, that is, functions f with $f\ \bot_{D_1} = \bot_{D_2}$.
- $\mathrm{strict}_D : (D_1 \to D_2) \to (D_1 \circ\!\!\to D_2)$ is the function that maps each continuous function to its strict counterpart.

3.5.2 Semantics of program schemes

Expressions such as $+\ \mathit{false}\ x$ are erroneous. They need a separate domain:

$$\mathit{Error} := \{\mathit{wrong}\}_\bot$$

These domains, together with a domain for functions, form the domain *Val* relevant for the meanings of lambda terms:

$$\begin{aligned} \mathit{Val} &= \mathbb{Z}_\bot \oplus \mathbb{T} \oplus \mathit{Fun} \oplus \mathit{Error} \\ \mathit{Fun} &= \mathit{Val} \to \mathit{Val} \end{aligned}$$

The case analysis construct function induces subdomain-specific variants $\mathrm{case}_S : \mathit{Val} \times (S \to \mathit{Val}) \to \mathit{Val}$. For $\mathbb{Z}_\bot$, it goes like this:

$$\mathrm{case}_{\mathbb{Z}_\bot}(x, f) := [f, \lambda b \in \mathbb{T}.\mathit{wrong}, \lambda g \in \mathit{Fun}.\mathit{wrong}, \lambda e \in \mathit{Error}.\mathit{wrong}](x).$$

For denotational semantics, an additional domain is required to hold *environments* which map variables to values:

$$\mathit{Env} := V \to \mathit{Val}$$

Furthermore, the semantics also deals with environments that map function names to values:

$$\mathit{FEnv} := F \to \mathit{Val}$$

For environments (both from *Env* and *FEnv*) the following special notation expresses the extension of an environment by a new binding:

$$f[x \mapsto v](z) := \begin{cases} v & \text{if } x = z \\ f(z) & \mathit{otherwise} \end{cases}$$

Note the similarity to the notation for substitutions in the lambda calculus.

For a denotational semantics for program schemes, assume a function $\alpha \in C \to \mathit{Val}$ which defines meanings for the constants. A fragment of it for $\mathbb{Z}_\bot$ and $\mathbb{T}$ may look as follows:

$$\begin{aligned} \alpha(n) &:= \mathrm{in}_{\mathbb{Z}_\bot}\, n \quad n \in \mathbb{Z} \\ \alpha(t) &:= \mathrm{in}_{\mathbb{T}}\, t \quad t \in \mathbb{T} \\ \alpha(+) &:= \mathrm{in}_{\mathit{Fun}}(\lambda x \in \mathit{Val}.\, \mathrm{in}_{\mathit{Fun}}(\lambda y \in \mathit{Val}.\, \mathrm{case}_{\mathbb{Z}_\bot}(x, \lambda x'.\, \mathrm{case}_{\mathbb{Z}_\bot}(y, \lambda y'.\, \mathrm{in}_{\mathbb{Z}_\bot}(x' + y'))))) \end{aligned}$$

The semantics function for terms $[\![\_]\!] : E' \to FEnv \to Env \to Val$ is defined by the following structural induction:

$$\begin{aligned}
[\![v]\!]\varphi\rho &:= \rho(v) \quad v \in V \\
[\![c]\!]\varphi\rho &:= \alpha(c) \quad c \in C \\
[\![f]\!]\varphi\rho &:= \varphi(f) \quad f \in F \\
[\![\lambda x.e]\!]\varphi\rho &:= \mathrm{in}_{Fun}(\lambda y \in Val.[\![e]\!]\varphi\rho[x \mapsto y]) \\
[\![(e_1\ e_2)]\!]\varphi\rho &:= \mathrm{case}_{Fun}([\![e_1]\!]\varphi\rho, \lambda g \in Fun.g([\![e_2]\!]\varphi\rho)) \\
[\![let\ v = e'\ in\ e]\!]\varphi\rho &:= [\![e]\!]\varphi\rho[v \mapsto [\![e']\!]\varphi\rho] \\
[\![if\ e\ then\ e_1\ e_2]\!]\varphi\rho &:= \mathrm{case}_{\mathbb{T}}([\![e]\!]\varphi\rho, \lambda b \in \mathbb{T}.b \to [\![e_1]\!]\varphi\rho, [\![e_2]\!]\varphi\rho)
\end{aligned}$$

The above semantics builds on an already-existing meaning function for the defined functions of a program scheme. Of course, this meaning function in turn results from an *application* of the above semantics. The main task to be done for the "lifting" of the expression semantics to program schemes is to describe the element of recursion: a fixpoint application does the trick. If S is the reference program scheme, $S(f)$ for $f \in F$ is to be the right-hand side e of the equation $f = e$ in the scheme. Here is how to compute a function environment for a program scheme S:

$$\Phi(S) := \mathrm{lfp}\,\lambda\varphi \in FEnv.\lambda f \in F.[\![S(f)]\!]\varphi\rho_{empty}$$

Now, it is straightforward to express the semantics of a whole program scheme. Let e_S be the body expression of the program scheme S, and ρ_{empty} an empty environment that maps each name to $\bot$.

$$[\![S]\!] := [\![e_S]\!](\Phi(S))\rho_{empty}$$

The semantics is easily converted into a strict variant by modifying the rules for function creation and *let*:

$$\begin{aligned}
[\![\lambda x.e]\!]\varphi\rho &:= \mathrm{in}_{Fun}(\mathrm{strict}(\lambda y \in Val.[\![e]\!]\varphi\rho[x \mapsto y])) \\
[\![let\ v = e'\ in\ e]\!]\varphi\rho &:= ([\![e']\!]\varphi\rho = \bot) \to \bot, [\![e]\!]\varphi\rho[v \mapsto [\![e']\!]\varphi\rho]
\end{aligned}$$

3.6 Lambda lifting

Notationally, program schemes are already very close to actual functional programs. Yet, their structure is sufficiently simple to make a formal description of their semantics simple and natural: Since explicit recursion can occur only at top level, the fixpoint computation factors out nicely to the outer level.

However, languages like Caml also allow for local recursive definition via the `let rec` construct. This construct also appears in Mini-Caml and its abstract syntax. Re-introducing it into program schemes would introduce numerous complications: The semantics would need to handle recursion at two different places, and the beauty of top-level recursion would be lost. Furthermore, the internal `let rec` is also tedious to implement correctly and efficiently, as it will turn out. Hence, proper translation of Mini-Caml into program schemes will require prior removal of `let rec` via a technique called *lambda lifting*

For the purposes of the discussion in this section, we will temporarily assume an additional *letrec* construct in the syntax:

⟨exp⟩ ::= *letrec* ⟨binding⟩ ⟨additional-binding⟩* *in* ⟨exp⟩
⟨binding⟩ ::= ⟨var⟩ = ⟨exp⟩
⟨additional-binding⟩ ::= *and* ⟨binding⟩

The right-hand side of a *letrec* must be an abstraction. This restriction is mainly technical, but has also found its way into the syntax of Caml.

The new *letrec* extends the reduction semantics for program schemes to also include *letrec*-bound identifiers in the scope of a redex. The idea of lambda lifting is to transform a program scheme with *letrec* into an equivalent one without one. Thus, it is a simple form of a *program transformation*, an important tool in compiler construction.

A good way to understand lambda lifting is to look at a few examples first and develop the general strategy from the insights gained on the way.

3.6.1 Strategies for removing *letrec*

Consider the following term involving *letrec* (ignoring for a moment that it does not make much sense operationally):

$$\mathit{let}\ i\ =\ 5\ \mathit{in}\ \mathit{letrec}\ f\ =\ \lambda x.f\ (+\ i\ i)\ \mathit{in}\ f\ (*\ i\ i)$$

Since this term involves "interior" recursion, and vanilla program schemes only provide top-level recursion, the basic approach to remove the *letrec* must be to "lift" f to the top level:

$$\begin{array}{l} f = \lambda x.f\ (+\ i\ i) \\ \mathit{let}\ i\ =\ 5\ \mathit{in}\ f\ (*\ i\ i) \end{array}$$

Unfortunately, the resulting program does not work because $\lambda x.f\ (+\ i\ i)$ has a free variable i, and the lifting has removed the term from the scope of i. Therefore, it is necessary to pass i as an additional parameter to f:

$$\begin{array}{l} f = \lambda i.\lambda x.f\ i\ (+\ i\ i) \\ \mathit{let}\ i\ =\ 5\ \mathit{in}\ f\ i\ (*\ i\ i) \end{array}$$

As an afterthought, it may have been more systematic to add the additional parameters first, and then lift them to the top level, thus avoiding an incorrect program at an intermediate stage:

$$\begin{array}{l} \mathit{let}\ i\ =\ 5\ \mathit{in}\ \mathit{letrec}\ f\ =\ \lambda x.f\ (+\ i\ i)\ \mathit{in}\ f\ (*\ i\ i) \\ \Longrightarrow \\ \mathit{let}\ i\ =\ 5\ \mathit{in}\ \mathit{letrec}\ f\ =\ \lambda i.\lambda x.f\ i\ (+\ i\ i)\ \mathit{in}\ f\ i\ (*\ i\ i) \\ \Longrightarrow \\ f\ =\ \lambda i.\lambda x.f\ i\ (+\ i\ i) \\ \mathit{let}\ i\ =\ 5\ \mathit{in}\ f\ i\ (*\ i\ i) \end{array}$$

It has now become visible that this introduction of additional parameters is really just the reverse of η reduction—so-called η *expansion*, and hence clearly a step which preserves the meaning of the program scheme. Also, pulling interior *letrec*-bound expressions to the top level is clearly compatible with β reduction and therefore semantics-preserving.

Unfortunately, it is not always quite sufficient to abstract over all free variables of a *letrec*-bound term. Consider the following, somewhat more involved example:

$$\begin{array}{l} \mathit{let}\ a\ =\ \ldots\ \mathit{in} \\ \mathit{let}\ b\ =\ \ldots\ \mathit{in} \\ \mathit{letrec}\ f\ =\ \lambda x.\ldots a\ldots g\ldots \\ \mathit{and}\ g\ =\ \lambda y.\ldots b\ldots f\ldots \\ \mathit{in}\ \ldots f\ldots g\ldots \end{array}$$

Here, a is free in f, and b is free in g. A naive application of the η expansion strategy produces the following result:

$$\begin{array}{l} \mathit{let}\ a\ =\ \ldots\ \mathit{in} \\ \mathit{let}\ b\ =\ \ldots\ \mathit{in} \\ \mathit{letrec}\ f\ =\ \lambda a.\lambda x.\ldots a\ldots g\ b\ldots \\ \mathit{and}\ g\ =\ \lambda b.\lambda y.\ldots b\ldots f\ a\ldots \\ \mathit{in}\ \ldots f\ a\ldots g\ b\ldots \end{array}$$

The result still has the same meaning as the original, but lifting is not possible yet: The η expansion has introduced new free variables in the bodies of f and g: b is now free in f, and a is free in g. Consequently, another application of the η expansion step is necessary:

$$\begin{array}{l} \mathit{let}\ a\ =\ \ldots\ \mathit{in} \\ \mathit{let}\ b\ =\ \ldots\ \mathit{in} \\ \mathit{letrec}\ f\ =\ \lambda b.\lambda a.\lambda x.\ldots a\ldots g\ a\ b\ldots \\ \mathit{and}\ g\ =\ \lambda a.\lambda b.\lambda y.\ldots b\ldots f\ b\ a\ldots \\ \mathit{in}\ \ldots f\ b\ a\ldots g\ a\ b\ldots \end{array}$$

Now, finally, the bodies both contain no more free variables, and lifting is possible:

$$\begin{array}{l} f\ =\ \lambda b.\lambda a.\lambda x.\ldots a\ldots g\ a\ b\ldots \\ g\ =\ \lambda a.\lambda b.\lambda y.\ldots b\ldots f\ b\ a\ldots \\ \mathit{let}\ a\ =\ \ldots\ \mathit{in} \\ \mathit{let}\ b\ =\ \ldots\ \mathit{in} \\ \ldots f\ b\ a\ldots g\ a\ b\ldots \end{array}$$

Unfortunately, there is no limit on how often the η expansion needs to be repeated until a fixpoint is reached. Obviously, performing this step successively repeatedly in a compiler is a costly matter just to cater to such a non-consequential language construct. Fortunately, a more systematic approach to the problem yields a more direct algorithm which is efficient enough in practice.

3.6.2 An algorithm for lambda lifting

The main goal for a lambda lifting algorithm is that is that it should transform the program only once. Hence, the algorithm needs to compute the set of variables over which abstraction is necessary. A straightforward approach to this is to formulate the constraints on these free variable sets as set equations and then solve those.

A prerequisite is that all identifiers in the program scheme must be unique: The program may bind no identifier twice. This simplifies the formulation of the algorithm, and generally prevents a few implementation headaches that have to do with inadvertent name capture problems.

The subject of the algorithm is a subterm e of a program scheme which has the following form:

$$\begin{array}{lrcl} \mathit{letrec} & f_1 & = & e_1 \\ \ldots & & & \\ \mathit{and} & f_n & = & e_n \\ \mathit{in} & e & & \end{array}$$

To prepare the f_i for lifting, abstraction is necessary over a set of variables A_{f_i}, and the following must hold:

$$A_{f_i} = \mathrm{free}(e_i) \cup \bigcup \{A_f \mid f \in \mathrm{free}(e_i), f\ \mathit{letrec}\text{-bound}\}$$

Using this equation and solving the resulting equation system over a whole program is still somewhat costly. However, it is possible to subdivide A_{f_i} and develop that into more manageable equations. There are

- the abstracted variables of all *letrec*-bound functions referenced from f_i bound in an outer scope, and
- the abstracted variables of all functions f_j bound in the same *letrec* as f_i itself.

The set of functions f_j referenced from f_i also receives a name:

$$m_{f_i} := \{f_j \in \text{free}(e_i)\}$$

Also, there are the functions referenced from f_i bound in some outer scope:

$$r_{f_i} := \{f \in \text{free}(e_i) \mid f \text{ } \textit{letrec}\text{-bound}\} \setminus m_{f_i}$$

Now, the revised equation for A_{f_i} goes as follows:

$$A_{f_i} = \text{free}(e_i) \cup \bigcup\{A_f \mid f \in r_{f_i}\} \cup \bigcup\{A_f \mid f \in m_{f_i}\}$$

In the revised equation, the recursion is limited to terms coming from right-hand sides of the same *letrec*. Since internal *letrec*s are usually not overly big, the resulting equation system is usually quite manageable. Also, the resulting equation system represents (for each *letrec* separately) a so-called *pure inclusion problem* for which efficient algorithms exist [DP82, WM92].

Chapter 4

Implementing the Lambda Calculus

The last chapter has provided the necessary machinery to translate Mini-Caml into a program schemes which are merely a straightforward extension to the lambda calculus. The language of program schemes is already considerably simplified compared to the abstract syntax of Mini-Caml. Implementing the lambda calculus means implementing Mini-Caml.

As regular and simple as the lambda calculus is, it is still very different from the machine language of real-world processors. Therefore, implementing the lambda calculus still involves a number of intermediate steps until it is suitable for native code generation. The first step is the derivation of an interpreter from the denotational semantics. From that follows an interpreter written in so-called *continuation-passing style* (or *CPS*) which is still visibly correct but closer to machine language. The CPS interpreter induces yet another intermediate language for programs, and a new sequence of interpreters implements CPS, each one getting closer to machine language.

4.1 Semantics into interpretation

The denotational semantics translates straightforwardly into an interpreter.

The output of the compiler phase described in the last chapter is an object of the type `scheme` in the `Lambda` structure:

```
type const =
    CInt of int
  | CString of string
  | CChar of char

type exp =
    Const of const
  | Ident of string
  | Builtin of string
  | Lambda of string * exp
  | Apply of exp * exp
  | Let of string * exp * exp
  | If of exp * exp * exp

type equation = string * exp
```

```
type scheme = equation list * exp
```

The datatype corresponding to *Val* is a sum type with identical structure except for a few additions:

```
type value =
    Unit
  | EmptyList
  | Int of int
  | Bool of bool
  | String of string
  | Char of char
  | Fun of (value -> value)
  | Cons of value * value
  | Wrong
```

Anticipating the demands of Mini-Caml, the `value` type contains a few more components than *Val*:

- `Unit` is the value of the unit constructor `()` in Caml, and `EmptyList` represents the empty list `[]`.
- `String` and `Char` are constructors for the corresponding constants in Caml.
- `Cons` is a constructor for pairs—aggregate data structures with two arbitrary components. In particular, they serve to implement lists.

Since the Mini-Caml abstract syntax distinguishes built-in identifiers and constants from normal ones, both of which are constants in the view of the denotational semantics. Therefore, the meaning function α has two counterparts in the implementation:

```
let eval_const c =
  match c with
    CInt i -> Int i
  | CString s -> String s
  | CChar c -> Char c
```

The meaning function for built-in identifiers is `eval_builtin`. The essential first-order constants are trivial to implement:

```
let eval_builtin i =
  match i with
    "()" -> Unit
  | "[]" -> EmptyList
  | "true" -> Bool true
  | "false" -> Bool false
```

Function constants must perform the same case analyses and injections as their counterparts in the semantics:

```
  | "inc" ->
      Fun
        (function x ->
          match x with
            Int x' -> Int (x' + 1)
          | _ -> Wrong)
```

```
| "dec" ->
    Fun
      (function x ->
        match x with
          Int x' -> Int (x' - 1)
        | _ -> Wrong)
| "zerop" ->
    Fun
      (function x ->
        match x with
          Int x' -> Bool (x' = 0)
        | _ -> Wrong)
| "+" ->
   Fun
      (function x ->
        Fun
          (function y ->
            match x with
              Int x' ->
                (match y with
                  Int y' -> Int (x' + y')
                |     _ -> Wrong)
            | _ -> Wrong))
```

Equality also comes in handy:

```
| "=" ->
    Fun (function x -> Fun (function y -> Bool (x = y)))
```

Also, identifiers named as in Caml are responsible for list construction and selection:

```
| "::" ->
    Fun
      (function x ->
        Fun
          (function y ->
            Cons (x, y)))
| "hd" ->
    Fun
      (function x ->
        match x with
          Cons (h, _) -> h
        | _ -> Wrong)
| "tl" ->
    Fun
      (function x ->
        match x with
          Cons (_, t) -> t
        | _ -> Wrong)
| _ -> Wrong
```

This list is necessarily incomplete and arbitrary.

Environments in the denotational semantics are functions, and it is natural to do the same in a functional programming language:

```
let empty_env x = raise Not_found
```

```
let extend_env env x y =
  function z ->
    if z = x
    then y
    else env z
```

Whereas the denotational semantics applies environments directly, the interpreter goes through a function `lookup_env` to do this. This makes it possible to change the representation for environments later into something more efficient without compromising the interpreter.

The semantics for expressions corresponds to a function `eval` which must have the expression to be evaluated, a function environment, and an environment. Again, the code corresponds closely to the semantics. Constants first:

```
let rec eval e fenv env =
  match e with
    Const c -> eval_const c
```

For identifiers, the interpreter must distinguish between locally bound identifiers, function names, and built-ins:

```
  | Ident i ->
      (try env i
      with Not_found -> fenv i)
  | Builtin b -> eval_builtin b
```

The rest is a one-to-one transliteration of the semantics:

```
  | Lambda (x, e) ->
      Fun
        (function y ->
          eval e fenv (extend_env env x y))
  | Apply (e1, e2) ->
      (match eval e1 fenv env with
        Fun f -> f (eval e2 fenv env)
      | _ -> Wrong)
  | Let (x, e', e) ->
      eval e fenv (extend_env env x (eval e' fenv env))
  | If (e, e1, e2) ->
      (match eval e fenv env with
        Bool t ->
          if t
          then eval e1 fenv env
          else eval e2 fenv env
      | _ -> Wrong)
```

The same holds true for the top-level evaluation function:

```
let eval_program (functions, e) =
  let rec fenv f =
    eval (List.assoc f functions) fenv empty_env
  in
  eval e fenv empty_env
```

4.2 Writing a definitional interpreter

This naive interpreter is surprisingly simple, mainly because the language being interpreted and the language the interpreter is written (the metalanguage) in are so similar. The correctness of the interpreter with respect to the denotational semantics seems obvious. However, there are a few crucial issues that require careful consideration before the above interpreter can be deemed correct:

1. Function application in the metalanguage follows the *call-by-value* evaluation strategy. Therefore, the interpreter also follows the call-by-value semantics, even though it is textually derived from the call-by-name semantics.

2. The interpreter handles parameter passing and return of the interpreted language by parameter passing and return of the metalanguage.

3. The interpreter handles binding by depending on the binding policy being the same in interpreted language and metalanguage. Would Caml use dynamic binding, so would the interpreted language.

4. The interpreter handles recursion with `let rec` rather than using an explicit fixpoint combinator. Hence, it uses the "implict" fixpoint operator of the metalanguage.

Consequently, even though the naive interpreter seems "obviously" correct, it makes use of quite a few more or less arbitrary properties of the metalanguage. Were the metalanguage to change its semantics, the language semantics would change as well. The interpreter is—on its own—not quite *definitional* yet as it delegates important semantics issues to the metalanguage rather than taking a stand on its own.

Of all the above issues, the second is the most important: Assembler programmers know that function application involves some fairly complicated machinery: it may store a "return address," create "stack frames," advance a "stack pointer" and jumps to a "target address." Thus, the interpreter shown above really fails to reveal a very important mechanism crucial to code generation at a later stage. As it turns out, resolving the procedure application issue also solves the first issue of pinning down an evaluation strategy. The binding policy as well as parameter passing will turn out to be easy to handle and is therefore left to a later section. The final issue of recursion is fortunately of little relevance as all of our intermediate languages—including machine code—possess an implicit recursion mechanism which a compiler can simply use.

The trick in making the interpreter definitional with respect to procedure application and return is to have it use a more primitive notion instead of the corresponding metalanguage constructs. Thus, the new interpreter will make two restrictive assumptions:

- The metalanguage allows only function applications that are tail calls.

- The metalanguage does not allow values to be returned from function applications.

These restrictions turn procedure applications into jumps with parameter passing. Both are mechanisms that translate straightforwardly into machine code.

The crucial idea in rewriting the interpreter to follow these restrictions is to recognize what actually happens with a value after the interpreter has computed it: That value is "passed" to some surrounding computation derived from its context. Look at the following expression

$$(f\ 2\ (g\ 23\ 17))$$

The interpreter computes—according to the rules of call-by-value evaluation—the value of (g 23 17) first, and then passes it to a computation corresponding to the evaluation context (f 2 [ ]). This context really represents the future of the computation of (g 23 17). In the naive interpreter, this future is always implicit somewhere in the folds of the implementation of the metalanguage. For effective compilation, it is necessary to make this context computation visible. It is called a *continuation*. In the terminology of the denotational semantics, it is a function $k : Val \to Val$, in this case $k = \lambda x.f\ 2\ x$.

The new interpreter will, instead of returning the value of an expression (a facility that is no longer available), pass it to a continuation corresponding to the context of that expression. This continuation (the *current continuation*) is an additional parameter to the new `eval` function. For the transition to happen, it is necessary to modify the `value` data type slightly: Since a function embedded in `value` will also no longer be able to return directly, it must take a continuation argument:

```
type value =
    Unit
  | EmptyList
  | Int of int
  | Bool of bool
  | String of string
  | Char of char
  | Fun of (value -> continuation -> value)
  | Cons of value * value
  | Wrong
and continuation = value -> value
```

Constants are evaluated as before:

```
let eval_const c =
  match c with
    CInt i -> Int i
  | CString s -> String s
  | CChar c -> Char c
```

The new `eval_builtin` function handles first-order constants the same as before:

```
let eval_builtin i =
  match i with
    "()" -> Unit
  | "[]" -> EmptyList
  | "true" -> Bool true
  | "false" -> Bool false
```

For function built-ins, the injected functions also need to accept a continuation argument and pass their results to it. The `inc` function is a unary example:

```
  | "inc" ->
     Fun
        (function x -> function k ->
          k
            (match x with
              Int x' -> Int (x' + 1)
            | _ -> Wrong))
```

The whole business is somewhat more tedious for the binary case:

```
  | "+" ->
     Fun
        (function x -> function k ->
          k (Fun
               (function y -> function k ->
                 k
                   (match x with
                     Int x' ->
                       (match y with
                         Int y' -> Int (x' + y')
                       | _ -> Wrong)
                   | _ -> Wrong))))
```

Environments are as before.

The new `eval` function takes a continuation argument:

```
let rec eval e fenv env k =
  match e with
```

The interpreter passes constants to the current continuation:

```
    Const c -> k (eval_const c)
```

Identifiers are handled the same as before:

```
  | Ident i ->
      k
        (try env i
        with Not_found -> fenv i)
  | Builtin b -> k (eval_builtin b)
```

The abstractions embedded into `value` need to accept a continuation argument:

```
  | Lambda (x, e) ->
      k (Fun
           (function y -> function k ->
             eval e fenv (extend_env env x y) k))
```

Correspondingly, application needs to provide suitable continuations so that it can process the values further:

```
  | Apply (e1, e2) ->
      eval
        e1 fenv env
        (function e1_val ->
          eval
            e2 fenv env
            (function e2_val ->
              match e1_val with
                Fun f -> f e2_val k
              | _ -> k Wrong))
```

The same holds true for `Let`:

```
  | Let (x, e', e) ->
      eval
        e' fenv env
        (function e'_val ->
          eval e fenv (extend_env env x e'_val) k)
```

... as well as for the conditional:

```
  | If (e, e1, e2) ->
      eval
        e fenv env
        (function e_val ->
          match e_val with
            Bool t ->
              if t
              then eval e1 fenv env k
              else eval e2 fenv env k
          | _ -> k Wrong)
```

Ultimately, the final return value of the program needs to be passed to a trivial identity continuation:

```
let eval_program (functions, e) =
  let rec fenv f =
    eval (List.assoc f functions) fenv empty_env (function v -> v)
  in
  eval e fenv empty_env (function v -> v)
```

The new interpreter uses a programming technique or style known as *continuation-passing style* or *CPS*. CPS is an important tool both in denotational semantics and programming pragmatics. It helps in this case because the CPS interpreter has the following properties:

1. The only non-tail calls in the program are either to metalanguage primitives such as `Int` or `match` or to functions like `eval_const` or `eval_builtin` which themselves contain no calls to other functions and thus could also be expanded into `eval`. None of them requires sophisticated machinery, as they call no other functions, and thus will typically be compiled "in-line" without involving function calls. The only function which ever "does" something on return is the inevitable identity continuation passed at top level. Of course, that continuation represents the empty context, and thus does nothing.

2. The code is suddenly "linear": Computations actually occur in the order in which they actually happen at interpretation time. In fact, this pins down the evaluation strategy: Even if the metalanguage were to use call-by-name, the CPS interpreter would still implement call-by-value semantics.

3. Every intermediate result has a name—the name of the continuation argument.

All these properties bring the CPS interpreter considerably closer to machine language than the naive version, and is also quite definitional already. Remember that machine language also has the following properties:

- Machine languages only have tail calls in the form of jumps. (Admittedly, CISCs usually have non-tail calls in the form of subroutine calls as well, but RISCs often do not.)

- Machine language is linear.

- Every intermediate result has a name, either the name of a register or of a storage location.

Still, the final objective of compilation is, of course, a compiler that *translates to* machine language rather than an interpreter *written in* machine language. In the compiler, therefore, it is necessary to translate our input program into CPS to make use of its useful properties.

4.3 Non-local exits

The CPS transformation has yet another pleasant side-effect: So far we have ignored the `raise` and `try` constructs of Mini-Caml in our translation of the lambda calculus. The pragmatic view to implementing this is to see `raise` and `try` as primitives. Since `try` is a special piece of it is necessary to transform

`try` e x `->` e'

into

`try (function () ->` e`) (function` x `->` e'`)`

and have `try` apply the thunk that represents e. With the direct-style interpreter, the only way of implementing `try` and `raise` is by using the corresponding constructs in the metalanguage. This, however, explains nothing, especially since the denotational semantics of the lambda calculus contains no provision for exceptions. In a definitional interpreter, therefore, this solution is not acceptable.

What does `try` do? It installs an *exception handler* which a subsequent `raise` invokes. Moreover, `raise` continues execution with the `try` construct. In other words: the continuation of `raise` is the continuation of the handler expression in the corresponding `try`. To properly handle this, the interpreter needs to propagate an additional continuation in addition to the current one: the *exception handler continuation*. The value domain needs to be extended once more:

```
type value =
    Unit
  | EmptyList
  | Int of int
  | Bool of bool
  | String of string
  | Char of char
  | Fun of (value -> handler -> continuation -> value)
  | Cons of value * value
  | Wrong
and continuation = value -> value
and handler = value -> value
```

The evaluation functions which previously accepted a current continuation argument now also accept a handler argument `h`:

```
let eval_builtin i =
  match i with
```

Naturally, all created functions must be extended. For example:

```
  | "inc" ->
      Fun
        (function x -> function h -> function k ->
          k
            (match x with
              Int x' -> Int (x' + 1)
            | _ -> Wrong))
```

Analagous changes propage though the central interpreter. These are all straightforward, however: The interesting cases are, of course, the primitive definitions for `try` and `raise` themselves:

```
  | "try" ->
      Fun
        (function thunk -> function h -> function k ->
          k (Fun
               (function handler -> function h -> function k ->
                 match thunk with
                   Fun thunk' ->
                     (match handler with
                       Fun handler' ->
                         thunk' Unit
                           (function exc ->
                             handler' exc h k)
                           k
                     | _ -> k Wrong)
                 | _ -> k Wrong)))
```

Note how the current continuation of the handler becomes the current continuation of the `try` application. The `raise` primitive just calls the handler, discarding the current continuation:

```
  | "raise" ->
      Fun
        (function exc -> function h -> function k ->
          h exc)
```

The `eval` and `eval_program` functions essentially stay as before except for the additional current handler `h` being passed around:

```
let rec eval e fenv env h k =
  match e with
    Const c -> k (eval_const c)
  | Ident i ->
      k
        (try env i
        with Not_found -> fenv i)
  | Builtin b -> k (eval_builtin b)
  | Lambda (x, e) ->
      k (Fun
           (function y -> function h -> function k ->
             eval e fenv (extend_env env x y) h k))
  | Apply (e1, e2) ->
      eval
        e1 fenv env
        h
        (function e1_val ->
          eval
            e2 fenv env
            h
            (function e2_val ->
              match e1_val with
                Fun f -> f e2_val h k
              | _ -> k Wrong))
  | Let (x, e', e) ->
      eval
        e' fenv env
```

```
        h
        (function e'_val ->
          eval e fenv (extend_env env x e'_val) h k)
  | If (e, e1, e2) ->
      eval
        e fenv env
        h
        (function e_val ->
          match e_val with
            Bool t ->
              if t
              then eval e1 fenv env h k
              else eval e2 fenv env h k
          | _ -> k Wrong)

exception Not_handled

let eval_program (functions, e) =
  let rec fenv f =
    eval
      (List.assoc f functions) fenv empty_env
      (function _ -> raise Not_handled)
      (function v -> v)
  in
  eval
    e fenv empty_env
    (function _ -> raise Not_handled)
    (function v -> v)
```

4.4 The CPS Transformation

Continuations have become so important in compiler construction that CPS warrants a closer look and more systematic study. Indeed, understanding CPS is a prerequisite to many newer research papers in compiler construction. At the heart of CPS is the *CPS transformation* which transforms a term in the lambda calculus into one using explicit continuations. This new representation for lambda terms (and, hence, the transformation) is exactly what is needed to reap the benefits of CPS as described in the previous section.

The presentation here follows the work by Danvy and Filinski [DF92].

For expository purposes, we will revert to the pure lambda calculus. All results will carry over smoothly to its applied variants. However, the notation of application (formerly $e_1\ e_2$) will change to $@e_1\ e_2$.

4.4.1 Classical CPS transformation

The central idea of the classical CPS transformation is by Plotkin and Fischer [Fis93, Plo75]. Recall that the interpreter used abstractions to represent continuations. The CPS transformation does the same.

4.1 Definition (Fischer/Plotkin Call-by-Value CPS Transformation)
It is a function $[\![\_]\!] : E \to E$*:*

$$\begin{aligned}
[\![x]\!] &:= \lambda k.@\ k\ x \\
[\![\lambda x.e]\!] &:= \lambda k.@\ k\ (\lambda x.[\![e]\!]) \\
[\![@\ e_1\ e_2]\!] &:= \lambda k.@\ [\![e_1]\!]\ (\lambda v_1.@\ [\![e_2]\!]\ (\lambda v_2.@\ (@\ v_1\ v_2)\ k)) \quad (v_1,\ v_2 \text{ fresh})
\end{aligned}$$

□

The Fischer/Plotkin CPS transformation is simple enough, and the following statement states its correctness with respect to call-by-value evaluation:

4.2 Theorem (Simulation)
Let e *be a lambda term. Let furthermore* eval_v *be call-by-value evaluation.*

$$\mathrm{eval}_v(e) = \mathrm{eval}_v(@\ [\![e]\!](\lambda x.x))$$

□

Moreover, the CPS transformation has a pleasant side effect:

4.3 Theorem (Indifference)
Let e *be a lambda term. Let furthermore* eval_v *be call-by-value evaluation and* eval_n *be call-by-name evaluation.*

$$\mathrm{eval}_n(@[\![e]\!](\lambda x.x)) = \mathrm{eval}_v(@[\![e]\!](\lambda x.x))$$

□

The consequence of the indifference property is that the CPS-transformed term is indifferent to the evaluation strategy: Call-by-name and call-by-value will produce the same result on a CPS term. Consequently, the CPS interpreter is now independent of the evaluation strategy of the metalanguage, which brings it a significant step closer to being definitional.

4.4.2 Avoiding administrative β redexes

Unfortunately, the Fischer/Plotkin CPS transformation is not suitable for direct application in realistic compilers: it produces humungous result terms. For example, the CPS version of $@(\lambda x.x)(@y\ y)$ is this:

$$\begin{aligned}
&\lambda k.@(\lambda k.@\ k(\lambda x.\lambda k.@k\ x)) \\
&\quad (\lambda m.@(\lambda k.@(\lambda k.@k\ y)(\lambda m.@(\lambda k.@k\ y)(\lambda n.@(@m\ n)\ k)))(\lambda n.@(@m\ n)\ k))
\end{aligned}$$

This term contains a large number of β redexes—in addition to the *beta* redex already present in the original term. Reducing those *administrative redexes* leads to the following, much more acceptable term:

$$\lambda k.@(y\ y)\ (\lambda a.@(@(\lambda x.\lambda k.(@k\ x))\ a)\ (\lambda a.@k\ a))$$

Hence, for practical intents and purposes, the Fischer/Plotkin CPS transformation needs to be accompanied by a post-reducer which removes the β reductions introduced by the vanilla transformation. This approach has the disadvantage that it still constructs the intermediate, large CPS term only to replace it immediately by something much smaller. It is much more desirable to compute the final result directly without large intermediate terms.

The method to achieve this "on-the-fly" post-reduction is to classify the abstractions and applications on the right-hand sides of the transformation into those

which will be part of an administrative β redex and those which will not. With the straightforward Fischer/Plotkin transformation, this is not possible—some abstractions and applications sometimes do take part in adminstrative redexes, and sometimes do not. However, it is possible to perform η expansion on the right-hand sides in a few, select instances, and then perform the classification.

The new transformation resulting from this has annotations on each λ and each @ indicating its classification: $\overline{\lambda}$ is for *static* abstractions that are part of administrative redexes (and therefore do not show up in the result term), and $\underline{\lambda}$ is for *dynamic* abstractions which definitely are part of the transformed term. Analogously, $\underline{@}$ denotes a dynamic application, and $\overline{@}$ a static one. The reformulation of the transformation is due to Danvy and Filinski [DF92], hence:

4.4 Definition (Danvy/Filinski CPS Transformation)
Let e be a lambda term. The Danvy/Filinski CPS Transformation *is a function $[\![\_]\!] : E \to (E \to E) \to E$:*

$$\begin{aligned}
[\![x]\!] &:= \overline{\lambda}\kappa.\overline{@}\kappa\, x \\
[\![\lambda x.e]\!] &:= \overline{\lambda}\kappa.\overline{@}\kappa(\underline{\lambda}x.\underline{\lambda}k.(\overline{@}[\![e]\!](\overline{\lambda}v.\underline{@}k\, v)) \\
[\![@e_1\, e_2]\!] &:= \overline{\lambda}\kappa.\overline{@}[\![e_1]\!]\,(\overline{\lambda}v_1.\overline{@}[\![e_2]\!]\,(\overline{\lambda}v_2.\underline{@}(\underline{@}v_1\, v_2)\,(\underline{\lambda}a.\overline{@}\kappa\, a)))
\end{aligned}$$

□

Note that, in this definition, κ stands for a continuation *at transformation time*; only k ever appears in the output of the transformation

The corresponding correctness statement is this:

4.5 Theorem
For a lambda term e, $\underline{\lambda}\kappa.\overline{@}[\![e]\!](\overline{\lambda}v.\underline{@}\kappa\, v)$ is $\beta\eta$-equivalent to the corresponding result term of the Fischer/Plotkin transformation.

□

The implementation of the Danvy/Filinski transformation is straightforward: Static abstractions and applications become the corresponding constructs in the metalanguage, and their dynamic counterparts become syntax constructors.

The introduction of additional η redexes allows for the classification of the applications and abstractions of a lambda term. Mostly, they participate in administrative redexes and therefore do not show up in the resulting term. However, there is an exception:

$$\overline{@}[\![\lambda f.@f\, x]\!](\overline{\lambda}v.v) = \lambda f.\lambda\kappa.@(@f\, x)\,(\lambda a.@\kappa\, a)$$

The residual term still contains an η redex introduced by the CPS transformation. This has potentially serious consequences, as the application in the original term is a tail call. Some modern programming languages based on the lambda calculus (most notably Scheme) demand that tail calls do not create new continuations. However, the above η redex is just that.

Therefore, it is necessary to augment the transformation to guard against this case. The idea is to duplicate the transformation rules: A new version of the rules is for "trivial" continuations of the form $\overline{\lambda}v.\underline{@}\kappa\, v$; the new rules avoid building the redex in the residual rules. The copied rules are called $[\![\_]\!]'$ in the following definition.

4.6 Definition (Tail-Recursive Danvy/Filinski CPS Transformation)
Let e be a lambda term. The tail-recursive Danvy/Filinski CPS Transformation *is*

a function $[\![\_]\!] : E \to (E \to E) \to E$*:*

$$\begin{aligned}
[\![x]\!] &:= \overline{\lambda}\kappa.\overline{@}\kappa\ x \\
[\![\lambda x.e]\!] &:= \overline{\lambda}\kappa.\overline{@}\kappa(\underline{\lambda}x.\underline{\lambda}k.(\overline{@}[\![e]\!]'k) \\
[\![@e_1\ e_2]\!] &:= \overline{\lambda}\kappa.\overline{@}[\![e_1]\!]\ (\overline{\lambda}v_1.\overline{@}[\![e_2]\!]\ (\overline{\lambda}v_2.\underline{@}(\underline{@}v_1\ v_2)\ (\underline{\lambda}a.\overline{@}\kappa\ a))) \\
\\
[\![x]\!]' &:= \overline{\lambda}k.\underline{@}k\ x \\
[\![\lambda x.e]\!]' &:= \overline{\lambda}k.\underline{@}k(\underline{\lambda}x.\underline{\lambda}k.(\overline{@}[\![e]\!]'k) \\
[\![@e_1\ e_2]\!] &:= \overline{\lambda}k.\overline{@}[\![e_1]\!]\ (\overline{\lambda}v_1.\overline{@}[\![e_2]\!]\ (\overline{\lambda}v_2.\underline{@}(\underline{@}v_1\ v_2)\ k))
\end{aligned}$$

□

Note that Theorem 4.5 requires a slight reformulation: The result of transforming a term e into CPS in a dynamic context is given by $\underline{\lambda}\kappa.\overline{@}[\![e]\!]'\kappa$.

4.5 Implementing the CPS Transformation

For implementing the CPS transformation, it is necessary to look at the form of the lambda terms generated by the transformation. It turns out that the transformation creates only a subset of all lambda terms; this subset lends itself to a specialized representation which makes implementing the transformation easier:

4.7 Theorem (Language of CPS terms)
The output of the tail-recursive Danvy/Filinski CPS transformation, $\underline{\lambda}\kappa.\overline{@}[\![E]\!]'\kappa$*, is exactly the language of the following grammar with start symbol* ⟨top-exp⟩*.*

⟨val-exp⟩ ::= ⟨const⟩ | ⟨var⟩ | λ ⟨var⟩ . λ κ . ⟨exp⟩
⟨exp⟩ ::= @ κ ⟨val-exp⟩
| *let* ⟨var⟩ = ⟨val-exp⟩ *in* ⟨exp⟩
| *if* ⟨val-exp⟩ *then* ⟨exp⟩ *else* ⟨exp⟩
| @ (@ ⟨val-exp⟩ ⟨val-exp⟩) (λ ⟨var⟩ . ⟨exp⟩)
| @ (@ ⟨val-exp⟩ ⟨val-exp⟩) κ
⟨top-exp⟩ ::= λ κ . ⟨exp⟩

□

Note that Theorem 4.7 is in light of a CPS transformation with the following rules for the applied lambda calculus:

$$\begin{aligned}
[\![\mathit{let}\ x\ = e'\ \mathit{in}\ e]\!] &:= \overline{\lambda}\kappa.\overline{@}[\![e']\!]\ (\overline{\lambda}v_1.\underline{\mathit{let}}\ x'\ =\ v_1\ \underline{\mathit{in}}\ \overline{@}[\![e[x \to x']]\!]\ \kappa) \\
[\![\mathit{if}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2]\!] &:= \overline{\lambda}\kappa.\overline{@}[\![e]\!]\ (\overline{\lambda}b.\underline{\mathit{if}}\ b\ \underline{\mathit{then}}\ \overline{@}[\![e_1]\!]\ \kappa\ \underline{\mathit{else}}\ \overline{@}[\![e_2]\!]\ \kappa)
\end{aligned}$$

This straightforward rule for *if* unfortunately has a pragmatic problem: The transformation propagates the context of an *if* term into both its branches, thereby duplicating it. For example, the lambda term

$$@f\ (\mathit{if}\ (\mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ z)\ \mathit{then}\ 4\ \mathit{else}\ 5)$$

results in a rather large CPS term:

$$\begin{aligned}
\lambda\kappa.\mathit{if}\ x\ \mathit{then}&(\mathit{if}\ y\ \mathit{then}\ @(@f\ 4)\ (\lambda v.@\kappa\ v)\ \mathit{else}\ @(@f\ 4)\ (\lambda v.@\kappa\ v)) \\
&\mathit{else}(\mathit{if}\ z\ \mathit{then}\ @(@f\ 4)\ (\lambda v.@\kappa\ v)\ \mathit{else}\ @(@f\ 4)\ (\lambda v.@\kappa\ v))
\end{aligned}$$

This excessive code duplication is undesirable in realistic compiler. It may therefore be preferable to cut off the context before moving onto the branches of an *if* expression:

$$[\![\mathit{if}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2]\!] := \overline{\lambda}\kappa.\underline{\mathit{let}}\ \kappa{=}\underline{\lambda} a.\overline{@}\kappa\ a \\ \underline{\mathit{in}}\ \overline{@}[\![e]\!]\ (\overline{\lambda} b.\underline{\mathit{if}}\ b\ \underline{\mathit{then}}\ \overline{@}[\![e_1]\!]'\ \kappa\ \underline{\mathit{else}}\ \overline{@}[\![e_2]\!]'\ \kappa)$$

With this rule in place, the CPS transformation is again "linear": The size of the output terms is linear in the size of the input terms. The new translation also introduces a new rule into the grammar for CPS terms:

$$\langle \mathrm{exp} \rangle\ ::=\ \mathit{let}\ \kappa\ =\ (\ \lambda\ \langle \mathrm{var} \rangle . \langle \mathrm{exp} \rangle\ )\ \mathit{in}\ \langle \mathrm{exp} \rangle$$

Now the machinery is in place to actually implement the transformation. The most straightforward method would be to translate `Lambda.exp` terms into `Lambda.exp` terms. However, this loses precious information from the CPS transformation: The residual terms do not distinguish between continuations and ordinary values, and it is difficult to special-case on the subset of `Lambda.exp` terms that the transformation produces. Theorem 4.7 provides the basis for a more specialized representation of CPS terms which goes into the interface of a structure named `Cps`:

```
type ident = string

type valexp =
    Const of Lambda.const
  | Ident of ident
  | Builtin of ident
  | Lambda of ident * ident * exp
and exp =
    Return of ident * valexp
  | Let of ident * valexp * exp
  | If of valexp * exp * exp
  | Call of valexp * valexp * cont
  | TailCall of valexp * valexp * ident
  | LetCont of ident * cont * exp
and cont = ident * exp
and cps = WithCont of ident * exp
and program = (ident * cps) list * cps
```

The transformation requires two identifiers, a unique constant one for continuations:

```
let ki = Rename.generate_unique "k"
```

... and a seed for fresh "normal" identifiers:

```
let ai = "a"
```

The implementation of the transformation proper very closely follows the Danvy/Filinski formulation. First, the counterpart for $[\![\_]\!]$:

```
let rec from_lambda_1 e k =
  match e with
    Lambda.Const c -> k (Const c)
  | Lambda.Ident i -> k (Ident i)
  | Lambda.Builtin b -> k (Builtin b)
  | Lambda.Lambda (x, e) ->
```

```
      k (Lambda (x, ki, from_lambda_2 e ki))
  | Lambda.Apply(e1, e2) ->
      from_lambda_1 e1
        (function v1 ->
         from_lambda_1 e2
            (function v2 ->
              let a = Rename.generate_unique ai in
              Call (v1, v2, (a, k (Ident a)))))
  | Lambda.Let(x, e', e) ->
      from_lambda_1 e'
        (function v' ->
          Let(x, v', from_lambda_1 e k))
  | Lambda.If(e, e1, e2) ->
      let a = Rename.generate_unique ai in
      LetCont(ki, (a, k (Ident a)),
              from_lambda_1 e
                (function v ->
                  If(v, from_lambda_2 e1 ki, from_lambda_2 e2 ki)))
```

Now $[\![\_]\!]'$:

```
and from_lambda_2 e ki =
  match e with
    Lambda.Const c -> Return (ki, Const c)
  | Lambda.Ident i -> Return (ki, Ident i)
  | Lambda.Builtin b -> Return (ki, Builtin b)
  | Lambda.Lambda (x, e) ->
      Return (ki, (Lambda (x, ki, from_lambda_2 e ki)))
  | Lambda.Apply(e1, e2) ->
      from_lambda_1 e1
        (function v1 ->
          from_lambda_1 e2
            (function v2 ->
              TailCall (v1, v2, ki)))
  | Lambda.Let(x, e', e) ->
      from_lambda_1 e'
        (function v' ->
          Let(x, v', from_lambda_2 e ki))
  | Lambda.If(e, e1, e2) ->
      from_lambda_1 e
        (function v ->
          If(v, from_lambda_2 e1 ki, from_lambda_2 e2 ki))
```

Two trivial functions take care of top-level expressions and program schemes:

```
let from_lambda_top e = WithCont(ki, from_lambda_2 e ki)

let from_lambda (equations, body) =
  (List.map
     (function (name, e) -> (name, from_lambda_top e))
     equations,
   from_lambda_top body)
```

4.6 Implementing CPS

Of course, the new representation requires a new interpreter formulated by specializing the original CPS interpreter for the standard lambda calculus. Thus, it reverts to explicitly passing around exception handlers and continuations. The value domain adds a `Cont` constructor:

```
type value =
    Unit
  | EmptyList
  | Int of int
  | Bool of bool
  | String of string
  | Char of char
  | Fun of (value -> handler -> continuation -> value)
  | Cont of continuation
  | Cons of value * value
  | Wrong
and continuation = value -> value
and handler = value -> value
```

The interpreter starts with constants:

```
let eval_const c =
  match c with
    Lambda.CInt i -> Int i
  | Lambda.CString s -> String s
  | Lambda.CChar c -> Char c
```

The `eval_builtin` function is the same as the one in the CPS interpreter for the ordinary lambda calculus as well as the functions for handling environments. The `eval` function is responsible for `Cps.exp` terms:

```
let rec eval e fenv (env : Cps.ident -> value) h =
  match e with
    Return (ki, ve) ->
      let v = eval_val ve fenv env in
      let Cont k = env ki in
      k v
  | Let (x, ve, e) ->
      eval e fenv (extend_env env x (eval_val ve fenv env)) h
  | If (ve, e1, e2) ->
      (match eval_val ve fenv env with
        Bool t ->
          if t
          then eval e1 fenv env h
          else eval e2 fenv env h)
  | Call(ve1, ve2, c) ->
      let v1 = eval_val ve1 fenv env in
      let v2 = eval_val ve2 fenv env in
      let Cont k = eval_cont c fenv env h in
      (match v1 with
        Fun f -> f v2 h k
      | _ -> k Wrong)
  | TailCall(ve1, ve2, ki) ->
      let v1 = eval_val ve1 fenv env in
```

```
      let v2 = eval_val ve2 fenv env in
      let Cont k = env ki in
      (match v1 with
        Fun f -> f v2 h k
      | _ -> k Wrong)
  | LetCont(ki, c, e) ->
      eval e fenv
        (extend_env env ki (eval_cont c fenv env h))
        h
```

The `eval_val` function handles terms of type `Cps.valexp`:

```
and eval_val ve fenv env =
  match ve with
    Const c -> eval_const c
  | Ident i ->
      (try env i
      with Not_found -> fenv i)
  | Builtin b -> eval_builtin b
  | Lambda(x, ki, e) ->
      Fun
        (function y -> function h -> function k ->
          eval e fenv
            (extend_env (extend_env env x y) ki (Cont k))
            h)
```

The `eval_cont` function handles terms of type `Cps.cont`:

```
and eval_cont (x, e) fenv env h =
  Cont
    (function v ->
      eval e fenv (extend_env env x v) h)
```

Finally, `eval_top` handles top-level expressions:

```
let eval_top t fenv =
  match t with
    WithCont (ki, e) ->
      eval e fenv
        (extend_env empty_env ki (Cont (function v -> v)))
        (function _ -> raise Not_found)
```

The new `eval_program` evaluates the right-hand sides of the program scheme equations prior to interpretating the body, to more closely model the semantics of Caml:

```
let eval_program (equations, body) =
  let rec fenv f =
    eval_top (List.assoc f equations) fenv
  in
  eval_top body fenv
```

4.7 Making Functions Machine-Friendly

Now that the representation for environments does not use *function* anymore, the next candidates for a representation change are the `Fun` values. The previous interpreters have all used the implementation of lexical scoping in the metalanguage to implement lexical scoping in the object language. Consequently, in order to make the interpreter more definitional, the `function` must go. Another look at the old `eval` clause for `Lambda` makes clear what the issues are:

```
  | Lambda(x, ki, e) ->
      Fun
        (function y -> function h -> function k ->
          eval e fenv
            (extend_env (extend_env env x y) ki (Cont k))
            h)
```

The value that `eval` returns for `Lambda` contains enough information to contain the application later on, or, more specifically, to evaluate the inner expression.

Obviously, evaluation of the inner expression requires values for `env` (the environment that was current at the time of creation of the `Fun` object), `x` (the identifier of the `Lambda` expression), `ki` (the identifier of the current continuation) `y` (the value that evaluation of `Apply` later passes), and the expression to be evaluated, `e`. Fortunately, `fenv` never changes, so it is not part of the object. The value for `y` only becomes known at application time, so `env`, `x`, `ki`, and `e` remain. An object containing the necessary information to perform an application is called a *closure*. Therefore, `value` receives a new constructor:

```
type value =
    Unit
  | EmptyList
  | Int of int
  | Bool of bool
  | String of string
  | Char of char
  | Fun of (value -> handler -> continuation -> value)
  | Closure of closure
  | Cont of continuation
  | Cons of value * value
  | Wrong
and closure = environment * ident * ident * exp
and environment = ident -> value
```

Functions are not the only values represented by meta-level functions: The same holds true for continuations and exception handlers. Witness `eval_cont` from the previous interpreter:

```
and eval_cont (x, e) fenv env h =
  Cont
    (function v ->
      eval e fenv (extend_env env x v) h)
```

Again, if continuations are to be packaged into non-function values, it is important to look at the values required by the body of the function representing the continuation: the environment `env`, the name of the intermediate result `x`, the exception handler, `h`, and the body of the continuation, `e`. This results in the following additional clause to the type definitions:

```
and continuation =
    Continuation of environment * handler * ident * exp
  | Stop
```

`Stop` is for representing the initial continuation `function v -> v` from the previous interpreter.

Similar thinking results in an analogous datatype for exception handlers:

```
and handler =
    Handler of environment * handler * ident * exp
  | Error
```

The `Error` handler is again for representing the initial handler, simply `raise Not_found` in the previous interpreter.

The `value` type still contains the old `Fun` constructor. The interpreter uses `Fun` for primitives, because most of do not have representations as `Lamdba` expressions.

The `eval` function starts off similarly to `eval` in the previous interpreter:

```
and eval e fenv env h =
  match e with
    Return (ki, ve) ->
      let v = eval_val ve fenv env in
      let Cont k = env ki in
      return k v fenv
  | Let (x, ve, e) ->
      eval e fenv (extend_env env x (eval_val ve fenv env)) h
  | If (ve, e1, e2) ->
      (match eval_val ve fenv env with
        Bool t ->
          if t
          then eval e1 fenv env h
          else eval e2 fenv env h)
```

The only difference is that the current-continuation parameter, `k`, is no longer a function—`eval` cannot call it directly but rather lets an auxiliary function called `return` handle this:

```
and return k v fenv =
  match k with
    Stop -> v
  | Continuation (env, h, x, e) ->
      eval e fenv (extend_env env x v) h
```

The new interpreter must handle `Closure` values in `Call` and `TailCall` expressions. The code that performs the procedure call is very similar to the code previously inside the function created for `Lambda` abstractions:

```
  | Call(ve1, ve2, c) ->
      let v1 = eval_val ve1 fenv env in
      let v2 = eval_val ve2 fenv env in
      let Cont k = eval_cont c fenv env h in
      (match v1 with
        Fun f -> f v2 h k
      | Closure (closure_env, x, ki, e) ->
          eval e fenv
            (extend_env (extend_env closure_env x v2) ki (Cont k))
            h
```

```
      | _ -> return k Wrong fenv)
  | TailCall(ve1, ve2, ki) ->
      let v1 = eval_val ve1 fenv env in
      let v2 = eval_val ve2 fenv env in
      let Cont k = env ki in
      (match v1 with
        Fun f -> f v2 h k
      | Closure (closure_env, x, ki, e) ->
          eval e fenv
            (extend_env (extend_env closure_env x v2) ki (Cont k))
            h
      | _ -> return k Wrong fenv)
```

The final clause, `LetCont`, is again as before:

```
  | LetCont(ki, c, e) ->
      eval e fenv
        (extend_env env ki (eval_cont c fenv env h))
        h
```

The `eval_val` function is also as before except for the `Lambda` case which gets simpler; all the work is now done in `eval` in the `Call` and `TailCall` clauses:

```
and eval_val ve fenv env =
  match ve with
    Const c -> eval_const c
  | Ident i ->
      (try env i
      with Not_found -> fenv i)
  | Builtin b -> eval_builtin b fenv
  | Lambda(x, ki, e) ->
      Closure (env, x, ki, e)
```

The same holds true for `eval_cont` whose work is mostly done by `return`:

```
and eval_cont (x, e) fenv env h =
  Cont (Continuation (env, h, x, e))
```

Now that continuations are no longer functions, it is necessary to change `eval_builtin` so it also calls `return` instead of `k` directly. Moreover, some primitives need to call `eval` which means that they need to supply an `fenv` parameter—hence, `fenv` must become an additional parameter for `eval_builtin`. Here are some of the simpler cases of `eval_builtin`:

```
let rec eval_builtin i fenv =
  match i with
    "()" -> Unit
  | "[]" -> EmptyList
  | "true" -> Bool true
  | "false" -> Bool false
  | "inc" ->
      Fun
        (function x -> function h -> function k ->
          return k
            (match x with
              Int x' -> Int (x' + 1)
            | _ -> Wrong)
```

```
          fenv)
| "+" ->
    Fun
      (function x -> function h -> function k ->
        return k
          (Fun
             (function y -> function h -> function k ->
               return k
                 (match x with
                   Int x' ->
                     (match y with
                       Int y' -> Int (x' + y')
                     |         _ -> Wrong)
                 | _ -> Wrong)
                 fenv))
          fenv)
```

Things become interesting with the implementations of `try` and `raise`. Again, some work shifts from `try` to `raise`:

```
| "try" ->
    Fun
      (function thunk -> function h -> function k ->
        return k
          (Fun
             (function handler -> function h -> function k ->
               match thunk with
                 Closure (env, x, ki, e) ->
                   (match handler with
                     Closure (handler_env, handler_x, handler_ki, handler_e) ->
                       eval e fenv
                         (extend_env (extend_env env x Unit) ki (Cont k))
                         (Handler (extend_env handler_env handler_ki (Cont k),
                                   h,
                                   handler_x, handler_e))
                   | _ -> return k Wrong fenv)
               | _ -> return k Wrong fenv))
          fenv)
| "raise" ->
    Fun
      (function exc -> function h -> function k ->
        match h with
          Handler (env, h, x, e) ->
            eval e fenv (extend_env env x exc) h
        | Error -> raise Not_found)
```

Chapter 5

The IBM POWER Architecture

The time has come to turn to actual machines in order to see what comes next in compilation. Most traditional compiler texts use imaginary processor architectures for that purpose to avoid the quirks and conventions associated with real-world microprocessors. However, compiling to a virtual processor gives few of the satisfactions associated with writing a compiler. Therefore, this chapter uses the IBM POWER/PowerPC architecture as a typical representative for modern RISC processor architectures. For now, the only concern is to understand the design principles underlying the architecture without concrete compilation issues in mind. As will become clear in the next chapter, however, the CPS representation of the last chapter converts neatly into real machine code.

5.1 Instruction Set Overview

IBM evolved the RS/6000 POWER architecture into the current PowerPC. The PowerPC mostly generalizes the older POWER chips. Therefore, the overlap is substantial, and the material presented here is common to both chips. Therefore, only the term "POWER" will be used.

The POWER architecture is a 32-bit RISC machine which has four separate functional units:

- the branch processor,
- the fixed-point processor,
- the floating-point processor,
- and the storage-control processor.

Each of these functional units executes a certain set of instructions and manages its own set of registers. Some are shared between several functional units, and special instructions move data between the register sets. Figure 5.1 shows the complete register set. Note that IBM in its documentation numbers bits in the exact opposite way as most other literature: Bit 0 is the most-significant bit, and the significance decreases with increasing bit indices. This chapter sticks with IBM's convention.

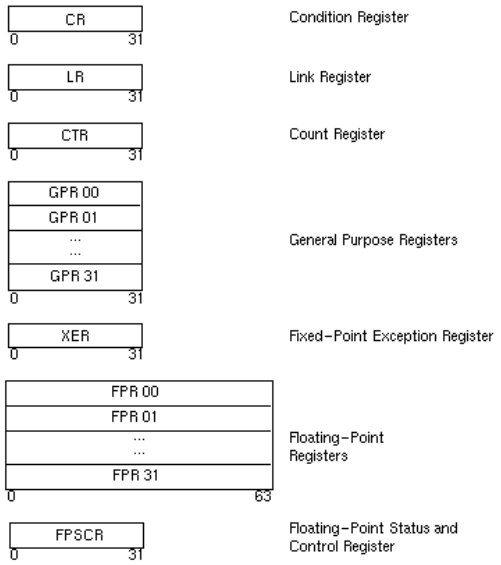


Figure 5.1: The POWER register set

5.1.1 The branch processor

is responsible for executing branches (conditional and unconditional), procedure calls, system calls, as well as certain logical operations. In addition to the condition register, the branch processor also manages the Link and Count registers.

The destination of a branch instruction can be one of the following:

- the sum of a constant and the address of the branch instruction itself,
- the absolute address given as an operand to the instruction,
- the content of the Link register, or
- the content of the Count register.

Certain branches store a return address into the link register, thereby supporting a form of procedure call. The Count register can serve as a loop count, and certain branch instructions implicitly decrement it and test if its value is zero.

The condition register is somewhat unusual compared to its counterpart in more traditional architectures: It consists of eight independent 4-bit condition fields with the following meanings for fixed-point instructions:

Bit 0 less than

Bit 1 greater than

Bit 2 equal

Bit 3 summary overflow

The summary overflow is irrelevant for the purposes of this chapter.

Each field of the condition register can be the target of selected fixed-point and floating-point instructions and can control branching. However, some instructions by default access fixed fields of the condition registers. It is possible to copy condition fields onto each other, thereby allowing the code to use secondary fields for backups.

```
bc    4,1,L38 # jump relative to L38 if not greater than in field 0
```

(In the instruction, 4 is the value of the "branch option field" which stands for "Branch if condition is false." The 1 is the number of the CR bit referenced. The `bc` instruction always refers to CR field 0.)

5.1.2 The fixed-point processor

The fixed-point processor manages the 32 32-bit *general-purpose registers* (GPRs) as well as a 32-bit fixed-point exception register. It implements loads (moves from memory *into* registers) and stores (moves *from* registers into memory), arithmetic, logical, compare, shift, rotate, trap, and system-control instructions. Some of its operations perform quite complex tasks and are reminiscent of older CISC architectures. The instruction format varies widely from instruction to instruction. Some instructions come in two variants, one of which has a mnemonic with a trailing dot `.` which indicates that the instruction sets the condition register field 0 according to the result of the operation.

Load and store The load instructions move information from a memory location into one of the GPRs. The store instructions do the reverse. Load and store instructions exist for bytes, halfwords, and words, but can only access word-aligned locations in memory. A load or store instruction denotes the memory location by an *effective address* which is either the contents of a GPR, the sum of the contents of a GPR and a signed 16-bit offset or the sum of the contents of two GPRs. Occasionally, GPR 0, when part of an effective address computation, actually denotes the number 0 rather than the contents of GPR 0.

```
lwz 6,16(5)  # [GPR 6] := [16 + [GPR 5]]
lhzx 6,5,4   # [GPR 6] := [[GPR 5] + [GPR 4]],
             # [GPR 6]_{0-15} := 0
stw 6,16(5)  # [16 + [GPR 5]] := [GPR 6]
```

Load and store with update Load and store instructions have an "update" form in which the base GPR is updated with the effective address in addition to the regular move of information from or to memory.

```
lwzu 6,16(5) # [GPR 6] := [16 + [GPR 5]], [GPR 5] := [GPR 5] + 16
stwu 6,16(5) # [16 + [GPR 5]] := [GPR 6], [GPR 5] := [GPR 5] + 16
```

String moves The string instructions allow the movement of data from storage to registers or from registers to storage regardless of alignment.

```
lswi 6,5,4  # [GPR 6] := <4 consecutive bytes at [GPR 5]>
stwi 6,5,4  # [[GPR 5]] := <4 consecutive bytes in [GPR 6]>
```

Arithmetic The fixed-point arithmetic instructions treat the contents of registers as 32-bit signed integers.

```
add 4,5,6   # [GPR 4] := [GPR 5] + [GPR 6]
add. 4,5,6  # [GPR 4] := [GPR 5] + [GPR 6], set CR0
```

Comparisons The compare instructions algebraically or logically compare the contents of a register with either a 16-bit immediate constant (signed or unsigned) or the contents of another register.

Algebraic comparison compares two signed integers. Logical comparison compares two unsigned integers.

```
cmp 1,0,4,5 # set CR1 according to [GPR 4] <cmp-algebraic> [GPR 5]
cmpl 1,0,4,5 # set CR1 according to [GPR 4] <cmp-logical> [GPR 5]
```

(The 0 field is a mandatory value without any special meaning.)

Logical operations Logical instructions perform bitwise logical operations.

```
andc 6,4,5 # [GPR 6] := [GPR 4] and ~[GPR 5]
```

Rotate and shift The rotate and shift operations work on the contents of a GPR in one of the following ways:

- The result of the rotation is inserted into the target register under the control of a mask. If the mask bit is 1, the associated bit of the rotated data is placed in the target register. If the mask bit is 0, the associated data bit in the target register is unchanged.
- The result of the rotation is ANDed with the mask before being placed into the target register.

The shift instructions logically perform left and right shifts. The result of a shift instruction is placed in the target register under the control of a generated mask.

```
rlmi 6,4,5,0,23 # [GPR 6] := [GPR 4] <rotate-left> [GPR 5]_{27-31}
                # ... but only bits 0-23
                # (rotate left then mask insert)
slw 6,4,5 # [GPR 6] := ([GPR 4] << [GPR 5]_{27-31}) & <mask>
```

(The `<mask>` in the `slw` instructions is controlled by bit 26 of GPR 5 through quite complicated rules.)

Move to/from special-purpose registers Several instructions move the contents of one Special-Purpose Register (SPR) into another SPR or into a GPR.

```
mfspr 6,1 # [GPR 6] := [EXC]
```

5.1.3 Extended Mnemonics

Many POWER instructions are excessively general, often hiding simple and common tasks behind complex operations. Because of this, IBM's assembler provides a large set of so-called *extended mnemonics* that are synonymous for special cases of complex statements. Some examples:

```
nop      # no-op, same as ori 0,0,0
mr 5,6   # (GPR 6) := (GPR 5), same as or 5,6,6
li 5,17  # (GPR 5) := 17, same as addi 5,0,17
```

5.2 Assembler Basics

An assembler source file consists of the following components:

Instructions consist of a mnemonic and, possibly, operands. An instruction corresponds to a machine instruction, and directsthe assembler to generate the corresponding executable code.

Labels are names that mark locations in the generated code.

Pseudo Operations are directives to the assembler independent of code generation per se.

An assembly source file consists of lines, each of which contains an instruction or a pseudo operation, optionally marked by a label:

⟨line⟩ ::= ⟨label⟩? ⟨statement⟩ ⟨comment⟩?
⟨statement⟩ ::= ⟨empty⟩
| ⟨mnemonic⟩ ⟨operands⟩?
| `.` ⟨pseudo-op⟩ ⟨pseudo-operands⟩?
⟨label⟩ ::= ⟨symbol⟩ `:`
⟨operands⟩ ::= ⟨operand⟩ ⟨another-operand⟩*
⟨another-operand⟩ ::= `,` ⟨operand⟩
⟨comment⟩ ::= `#` ⟨char⟩*
⟨pseudo-operands⟩ ::= ⟨pseudo-operand⟩ ⟨another-pseudo-operand⟩*
⟨another-pseudo-operand⟩ ::= `,` ⟨pseudo-operand⟩

As the assembler advances through a source file, it generates code and keeps a *location counter* marking the address of the code generated. The symbol `$` refers to the value of the location counter. Also, certain pseudo operations change the location counter directly without generating any code.

5.2.1 Pseudo Operations

Here are some common pseudo operations:

⟨statement⟩ ::= `.set` ⟨symbol⟩ `,` ⟨expression⟩

makes ⟨symbol⟩ a synonym for the value of ⟨expression⟩.

```
.set ap,14
lil ap,2   # assembler substitutes value 14 for the symbol
           # Note that ap is a register number as lil's operand.
```

⟨statement⟩ ::= `.byte` ⟨expressions⟩?
⟨expressions⟩ ::= ⟨expression⟩ ⟨another-expression⟩*
⟨another-expression⟩ ::= `,` ⟨expression⟩

generates consecutive bytes corresponding to the values of the parameters of `.byte`.

```
.byte 0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x20,0x77,0x6F,0x72,0x6C,0x64
.byte "Hello, world"
.byte 'H,'e,'l,'l,'o,',,' ,'w,'o,'r,'l,'d
```

There are corresponding pseudo operations `.double`, `.float`, `.long`, `.short`, `.string`, and `.vbyte`.

⟨statement⟩ ::= `.align` ⟨expression⟩

advances the current location counter until a boundary specified by the parameter is reached. The parameter values correspond to the following alignments:

0 byte

1 halfword (16 bits)

2 word

3 doubleword

```
.byte   1 # Location counter now at odd number
.align  1 # Location counter is now at the next halfword boundary.
.byte   3,4
```

5.2.2 AIX conventions

Knowing the machine language and assembler syntax of an architecture is not enough to be able to write running programs: The operating system and runtime environment impose a number of constraints and conventions on assembler code. In the case of AIX, these conventions have to do with the subdivision of a program into sections and procedure calling conventions.

Programming with the TOC An AIX executable consists of *sections* that contain different parts of the running program. The important sections are called `.text`, `.data`, and `.bss`:

.text contains code or read-only data.

.data contains read-write data.

.bss contains uninitialized mapped data.

Each section consists of subsections called *csects*. A csect is assigned to a *storage mapping class* further describing the role of a csect. The set of storage mapping classes is fixed, and each class has a two-letter name. The storage mapping class also determines a certain section. For example, `PR` is for executable code residing in the `.text` section, and `RO` is for read-only data also residing in `.text`

The code for each code-generating statement of an assembler source file is assigned to a specific csect, and assembler and linker together take care to assemble the various csects into consecutive chunks of code.

Code can be assigned to a csect with the `.csect` directive. For example,

```
.csect bla[pr]
```

announces that subsequent code will be put into a csect named `bla[pr]` with (obviously) storage mapping class `PR`.

Now, since instruction opcodes are all exactly 32 bits in size, they cannot contain a full absolute memory reference. Instead, memory access needs to be indirect. Therefore, AIX provides access to absolute memory locations via a *table of contents*, or *TOC* accessible through a GPR. The TOC is simply a table of indirections

accessible globally in a program. It can contain data directly or pointers to csects. Only the latter case is of interest here.

TOC entries have special storage mapping classes. A TOC entry with storage mapping class `TC` contains the address of a csect or a global symbol. The `.toc` directive announces that subsequent code goes into the TOC, and the `.tc` directive creates TOC entries. As a convention, AIX programs keep a pointer to the TOC in general-purpose register 2. Here is a usage example:

```
        .set    RTOC,2
        .csect prog1[pr]
        ...
        l  5,TCA(RTOC)          # [GPR5] := a[rw]
        ...
        .toc
TCA:    .tc  a[tc],a[rw]        # csect a[rw] goes into TOC name a[tc]

        .csect a[rw]
        .long 25
```

Calling Conventions AIX also specifies a set of *calling conventions* meant to ease interoperability between different programming languages. It is not the main concern of the compilers here, but since all programs run in the same environment, they must at least outwardly observe these conventions.

Register	Status	Use
GPR0	volatile	In function prologs.
GPR1	dedicated	Stack pointer.
GPR2	dedicated	Table of Contents (TOC) pointer.
GPR3	volatile	First word of a function's argument list; first word of a scalar function return.
GPR4	volatile	Second word of a function's argument list; second word of a scalar function return.
GPR5	volatile	Third word of a function's argument list.
GPR6	volatile	Fourth word of a function's argument list.
GPR7	volatile	Fifth word of a function's argument list.
GPR8	volatile	Sixth word of a function's argument list.
GPR9	volatile	Seventh word of a function's argument list.
GPR10	volatile	Eighth word of a function's argument list.
GPR11	volatile	In calls by pointer and as an environment pointer for languages that require it (for example, PASCAL).
GPR12	volatile	For special exception handling required by certain languages and in glink code.
GPR13:GPR31	nonvolatile	These registers must be preserved across a function call.

Figure 5.2: General Purpose Register Conventions

The preferred method of using GPRs is to use the volatile registers first. Next, use the nonvolatile registers in descending order, starting with GPR31 and proceeding down to GPR13. GPR1 and GPR2 must be dedicated as stack and Table of Contents (TOC) area pointers, respectively. GPR1 and GPR2 must appear to be

saved across a call, and must have the same values at return as when the call was made.

Chapter 6

Data Representation

Before machine code generation is feasible, it is necessary to address a number of run-time issues that are independent of code generation per se, but are needed for code to run. Most of these issues did not become evident in the interpreter as they implicitly use the run-time environment of the metalanguage which is sufficiently powerful. With assembly language, however, the runtime environment is almost non-existent, so it is necessary to add a few ingredients.

6.1 Addressing Run Time Issues

The interpreters for the lambda calculus and the CPS language—despite being definitional—implicitly make use of a number of facilities of the metalanguage. They have gone unnoticed so far because they are not even part of the denotational semantics: the metalanguage of mathematics either provides these facilities as well or is simply not concerned with the operational consequences of one or the other implementation of them.

Among these issues are:

Data representation How does a value from a sum domain translate into the domain of bits and bytes?

Memory management In mathematics, "memory" is infinite—it does not matter how many objects a running program "creates." Not so for real machines.

Of these issues, data representation and memory management are very closely related and pose important design constraints on the other two. Therefore, they are first.

6.1.1 Data representation and memory management

From the interpreters, it is not obvious what the exact requirements concerning data representation on the actual machine are. A naive coding of the components of the `value` data type would look like this:

`Unit` The word 0.

`EmptyList` The word 0.

`Int` A word with the same value as the integer argument.

`Bool` The word 0 for `true`, the word 1 for `false`.

`Char` A byte with the ASCII of the character.

`String` An area in memory (the *heap*) where the first word contains the length of the string and remaining bytes contain the characters.

`Closure` An area in memory where the first word is the address of the closure environment, and the second word is the address of the code representing the body of the abstraction that created the closure.

`Cons` An area in memory which begins with the first (head) component of the pair, after which comes the second (tail) component of the pair.

`Wrong` The word 0.

Unfortunately, the naive coding is unsuitable for several reasons. For illustration, consider the following Mini-Caml program:

```
let pair x = [x]

let main () = begin pair 42; pair [42] end
```

The program applies `pair` once to a number and once to a list. Whereas the number fits in one word, the list—really a pair—does not. However, `pair` needs to handle its argument in some way to stuff it into a pair. Unfortunately, since it is polymorphic, it has no information about the type of the object and therefore cannot determine how to put it inside a pair. Intuitively, the type of the object does not matter since `pair` does not "look" at the object but only puts it somewhere else. However, it at least needs to know how *big* the object is.

The obvious solution is to make all objects have the same size—a single word comes to mind. Some objects already fit in one word. Those which do not are composite objects easily represented by a pointer to a memory area representing the object rather than the memory area itself. The process of replacing a composite object by a pointer to it is called *boxing*, the reverse—dereferencing the pointer—it called *unboxing*.

However, the resulting representation still has two problems:

```
let main () = fst 42
```

The semantics of the above program is obviously undefined—it contains a type error. However, the data representation does not reflect that fact: The representation for the number `42` looks just like a pointer to a pair, and the program, when run, will happily dereference it, leading to a more or less random result. Even though C programs routinely exhibit random behavior because of exactly such errors, this is just as clearly undesirable. If pointers were distinguishable from integers, the running program could check that the argument to `fst` is indeed a pointer to a pair, and, on failure, report the error in an orderly fashion which allows the programmer to fix the bug.

In the full Caml language, the type system prevents such errors. Therefore, run-time type checks are not necessary there. Conceivably, a static type system with a suitable checking machine could be added to the compiler. However, even with static type checking in place, another problem remains which is not as easily moved to the realm of the static:

```
let rec garbage count =
  if count = 0
  then 42
  else begin [42]; garbage (count - 1) end;

let main () = garbage 99999999
```

Apart from the fact that the program does not do anything useful: It creates 99999999 pairs when it is run, and each pair consumes two words of memory. Most realistic machines do not have that much memory. However, the pair `[42]` becomes "garbage" immediately after its creation: It is no longer accessible for the program (it is *dead*), and therefore the memory that it takes up can immediately be re-used for other purposes. However, Mini-Caml has no equivalent to Pascal's `dispose` directive or C's `free` function to make the memory available again. Also, even though it is clearly statically visible in the example when the pair dies, such a static analysis is usually much harder in realistic programs. (Even though it is possible.)

Therefore, most implementations of high-level languages provide a facility called *garbage collection* or just plain *GC*. When a program runs out of memory, it calls the GC which reclaims the memory of all objects which the running program may no longer access. GC techniques and algorithms fill a book. However, it is clear that the GC needs support from the language implementation to do its work:

1. It needs access to all objects that the running program has "immediate" access to. In our interpreter, this would be the values of the registers. These objects form the *root set.*

2. The GC needs to trace through all objects reachable—directly or indirectly—from the root set to form the set of *live objects*. To do this, it needs to distinguish *immediate* objects which fit in a word from pointers, and it needs to know what components of composite objects are reachable through the object.

The first point will be easy to satisfy. However, the second requirement really means that the objects have to carry a limited amount of type information for the tracing to be possible.

The next draft of the data representation could look like this:

- All values that the running program handles are pointers to objects in the heap.

- The first word of an object in the heap either represents a small immediate object (the empty list, unit, true, false, or a character) or the type of larger object.

- The second word of an object contains its size in bytes. This could be inferred from the type in most cases, but not for strings, for instance. Generally, it is a useful piece of information to have around.

- The remaining words contain the actual data representing the object.

This representation fulfills all the requirements. However, it is still inefficient: For every integer operation, a dereference becomes necessary because all integers reside in the heap. Moreover, the result must be stored in the heap, resulting in further overhead. However, on 32-bit architectures, pointers always have the following form (again reverting to conventional bit counting):

Bits 31–2	Bits 1–0
x	00

The two least significant bits are always 0 and therefore contain no useful information. If 30 bits are sufficient, words of the following format (*descriptors*) could represent objects:

Type	Bits 31–2	Bits 1–0
Integer	integer value	00
Unit	00000000000000000000000000000000	01
Empty list	00000000000000000000000000000001	01
False	00000000000000000000000000000010	01
True	00000000000000000000000000000110	01
Character	00000000000000000000xxxxxxxx11	01
Heap object	upper bits of pointer	10

The lower two bits encoding partial type information are called the *tag* of the object. Note that choosing a tag of 00 for integers means that addition and subtraction work just as before, and only division and multiplications require shifts for the necessary adjustments. The fact that pointers no longer represent themselves is easily remedied by an offset of -2 in indirect addresses—something that modern processors provide anyway. One tag value is still available for miscellaneous purposes.

An object in the heap has the following layout—starting with the address pointed to by the descriptor:

Word 0	Word 1	...
Header	Data	...

The header has the following layout:

Bits 31–4	Bits 3-0
Size (bytes)	Type

The type tag in the header can have the following values:

Type	Tag
Pair	0000
Closure	0001
Continuation	0011
Environment	0100
String	1000

The assumption is that every heap object with a type tag that has its upper bit sets is actually a *bitmap* which does not contain further descriptors, but instead some opaque data of no interest to, say, the garbage collector. Again, the encoding is a matter of convention, not necessity.

Note also that no type tag has a trailing 10 bit pattern—this is reserved for representing broken hearts in a moving garbagce collector.

6.1.2 Environments and continuations

For first-order heap objects such as strings or pairs, it is straightforward to find a suitable heap representation. For environments and continuations, slightly more care is needed because both carry identifiers—at least as represented in the interpreters. However, in Mini-Caml, identifier usage and binding follows the rules of *static scoping*. The name suggests that a compiler can leave identifiers in the static realm and drop them for execution purposes.

The first-order representation of environments indeed suggests that environment access is possible through a simple index. As the interpreter creates new bindings in a LIFO fashion, the index is really a reverse stack depth. A possible layout for environment objects is therefore the following:

Word 0	Word 1	Word 2	...
Header	previous environment	Entry 0	further entries

In this setup, the environment consists of linked *frames*, each of which can contain any number of bindings. Note that currently, binding works in such a way that bindings always happen one variable at a type; frames therefore typically contain only one binding each.

With environments out of the way, closures are easy:

Word 0	Word 1	Word 2
Header	Environment	Code

Of course, the table leaves open what "Code" really means. Supposedly, it could just be a pointer to the machine code represented by the closure. This would, however, destroy the uniformity of heap object layout: A heap object is supposed to consist of descriptors only. Therefore, it is better to have "Code" be a *code template* which in turn contains a reference to a *code vector* which contains bitmap data only

Type	Tag
Code vector	`1001`
Code template	`0101`

A closure contains a descriptor pointing to the template.

Continuations and exception handlers now are just as straightforward:

Word 0	Word 1	Word 2	Word 3	Word 4
Header	Environment	Continuation	Exception handler	Code template

Appendix A

Auxiliary functions

By the time of this writing, a number of useful functions is not provided by the Ocaml language. In this chapter, we define some modules containing the functions needed in this text.

A.1 Listplaus

The module `Listplus` provides auxiliary functions on lists.

A.1.1 Taking lists apart

```
let rec drop n xs =
  if n = 0 then
    xs
  else
    match xs with
      x :: xs -> drop (n - 1) xs

let rec take n xs =
  if n = 0 then
    []
  else
    match xs with
      [] -> []
    | x :: xs -> x :: take (n - 1) xs

let rec update xs n y =
  match xs with
    [] -> []
  | x::xs ->
      if n = 0 then
        y::xs
      else
        x::update xs (n-1) y

let null xs =
  match xs with
    [] -> true
  | _ -> false
```

A.1.2 Searching

```
let position y xs =
  let rec f i xs =
    match xs with
      [] -> None
    | x::xs ->
        if x = y then Some i else f (i+1) xs
  in
  f 0 xs
```

A.1.3 Filtering

```
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 []
```

A.1.4 Sets as lists with unique elements

```
let insert set x = x :: set

let empty_set = []

let union l1 l2 =
  List.fold_left
    (function l -> function e ->
      if List.mem e l2 then l else e::l)
    l2 l1

let union_list l = List.fold_left union [] l

let intersection (l1 : Ident.t list) l2 =
  filter (function x -> List.mem x l2) l1

let difference l1 l2 =
  filter (function e -> not (List.mem e l2)) l1
```

A.1.5 Unclassified

```
let from_to n1 n2 =
  let rec loop n result =
    if n > n2
    then List.rev result
    else loop (n + 1) (n::result)
  in loop n1 []
```

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Boc76] G. V. Bochmann. Semantic evaluation from left to right. *Communications of the ACM*, 19:55–62, 1976.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[Cha87] Nigel P. Chapman. *LR parsing: theory and practice*. Cambridge University Press, Cambridge, UK, 1987.

[DeR69] Franklin L. DeRemer. *Practical Translators for LR(k) Parsers*. PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Ma., 1969.

[DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, 1971.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.

[DP82] Franklin DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1982.

[DS95] Charles Donnelly and Richard Stallman. *Bison—The YACC-compatible Parser Generator*. Free Software Foundation, Boston, MA, November 1995. Part of the Bison distribution.

[Fis93] Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):259–288, 1993.

[Ive86] Fred Ives. Unifying view of recent LALR(1) lookahead set algorithms. *SIGPLAN Notices*, 21(7):131–135, July 1986. Proceedings of the SIGPLAN'86 Symposium on Compiler Construction.

[Ive87a] Fred Ives. An LALR(1) lookahead set algorithm. Unpublished manuscript, 1987.

[Ive87b] Fred Ives. Response to remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(8):99–104, August 1987.

[Joh75] S. C. Johnson. Yacc—yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[Lee93] Rene Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic Publishers, Boston, 1993.

[PC87] Joseph C.H. Park and Kwang-Moo Choe. Remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(4):30–32, April 1987.

[PCC85] Joseph C. H. Park, K. M. Choe, and C. H. Chang. A new analysis of LALR formalisms. *ACM Transactions on Programming Languages and Systems*, 7(1):159–175, January 1985.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Spe94] Michael Sperber. Attribute-directed functional LR parsing. Unpublished manuscript, October 1994.

[SSS90] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*, volume II (LR(k) and LL(k) Parsing) of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1990.

[ST95] Michael Sperber and Peter Thiemann. The essence of LR parsing. In William Scherlis, editor, *Proceedings ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 146–155, La Jolla, CA, USA, June 1995. ACM Press.

[ST00] Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(2):224–264, March 2000.

[WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau — Theorie, Konstruktion, Generierung*. Lehrbuch. Springer-Verlag, Berlin, Heidelberg, 1992.