

# Systematic Compiler Construction

Michael Sperber      Peter Thiemann

November 20, 2003

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

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

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

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



## Chapter 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  $A \rightarrow \alpha$  for a nonterminal  $A$  and a sequence  $\alpha$  of grammar symbols.

$\epsilon$  is the empty sequence;  $|\xi|$  is the length of sequence  $\xi$ . Furthermore,  $\alpha^k$  denotes a sequence with  $k$  copies of  $\alpha$ , and  $\xi|_k$  is the sequence consisting of the first  $k$  terminals in  $\xi$ . □

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

$$\alpha \Rightarrow \beta \Leftrightarrow \alpha = \delta A \gamma \wedge \beta = \delta \mu \gamma \wedge A \rightarrow \mu$$

and  $\overset{*}{\Rightarrow}$  denotes the reflexive and transitive closure of  $\Rightarrow$ . A derivation from  $\alpha_0$  to  $\alpha_n$  is a sequence  $\alpha_0, \alpha_1, \dots, \alpha_n$  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 accepts 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  $[X] : T^* \rightarrow \mathcal{P}(T^*)$  which fulfills the following equation:

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

Hence, a sequence of terminals  $\xi$  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:

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

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

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

Thus fortified, the definition for nonterminals follows:

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

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:

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

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— $\text{first}_k$  and  $\text{follow}_k$ .

### 2.3 Definition ( $\text{first}_k$ , $\text{follow}_k$ )

For an integer  $k$ ,  $\text{first}_k$  and  $\text{follow}_k$  are defined as follows:

$$\begin{aligned} \text{first}_k &: V^* \rightarrow \mathcal{P}(T^{\leq k}) \\ \text{first}_k(\alpha) &:= \{ \xi|_k \mid \alpha \xRightarrow{*} \xi \} \\ \text{follow}_k &: N \rightarrow \mathcal{P}(T^k) \\ \text{follow}_k(A) &:= \{ \xi \mid S \xRightarrow{*} \beta A \gamma \wedge \xi \in \text{first}_k(\gamma) \} \end{aligned}$$

□

The  $\text{first}_k$  function computes, for a sequence of grammar symbols  $\alpha$ , all  $k$ -sequences of terminals which might result from a derivation starting with  $\alpha$ . 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:

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

□

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

$$[A](\xi) = \bigcup_{\substack{A \rightarrow \alpha \\ \xi|_k \in (\text{first}_k(\alpha) \text{ follow}_k(A))|_k}} [\alpha](\xi)$$

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  $A \rightarrow \alpha$  is computed as follows:

$$\text{LLA}_k(A \rightarrow \alpha) := (\text{first}_k(\alpha) \text{ follow}_k(A))|_k$$

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

$$\text{LLA}_k(A \rightarrow \alpha) \cap \text{LLA}_k(A \rightarrow \beta) = \emptyset.$$

□

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 `L1` 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  $S \rightarrow A$ .

□

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. The lookahead has the same purpose here as for recursive-descent parsers: to make parsing deterministic by describing which input symbols may come next. As a recursive-ascent parser needs to choose a rule only after having seen the whole right-hand side of a production, an recursive-ascent lookahead consists of the symbols that follow the non-terminal on the left-hand side of a production. A formal derivation will follow soon.

#### 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  $A \rightarrow \alpha \cdot \beta$  ( $\rho$ ) where the dot indicates the position, and  $\rho$  is the lookahead. If the lookahead is not used (or  $k = 0$ ), it is omitted. A kernel item has the form  $A \rightarrow \alpha \cdot \beta$  ( $\rho$ ) with  $|\alpha| > 0$ . A predict item has the form  $A \rightarrow \cdot \alpha$  ( $\rho$ ).

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

□

Whereas the  $\llbracket \cdot \rrbracket$  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  $q_0 = \{S \rightarrow \cdot A\}$ .

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:

$$\text{predict}(q) := \left\{ B \rightarrow \cdot \nu (\tau) \mid A \rightarrow \alpha \cdot \beta (\rho) \Downarrow^+ B \rightarrow \cdot \nu (\tau) \right. \\ \left. \text{for } A \rightarrow \alpha \cdot \beta (\rho) \in q \right\}$$

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

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

The relation  $\Downarrow$  also shows how to compute the lookahead of an item as the concatenation of the symbols that may follow the non-terminal on the left-hand side and the lookahead of the original item.

The union of  $q$  and  $\text{predict}(q)$  is called the closure of  $q$ . Henceforth,

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

denotes the closure of a state  $q$ .

For a state  $q$  and a grammar symbol  $X$ :

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

□

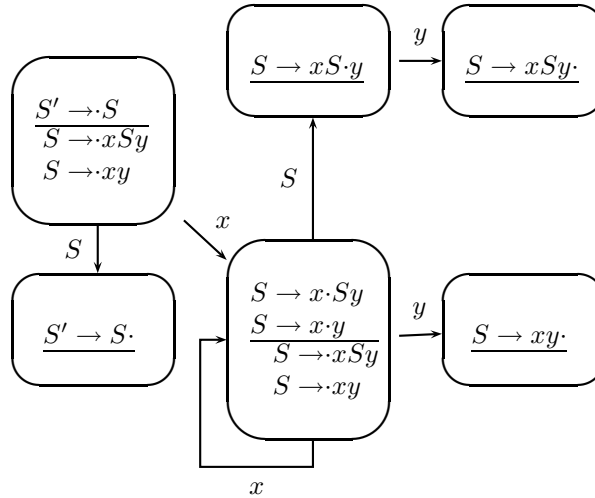


Figure 2.2: LR state diagram for  $S' \rightarrow S, S \rightarrow xSy, S \rightarrow xy$

The predict items of a state  $q$  are predictions on what derivations the parser may enter next when in state  $q$ . The elements of  $\text{predict}(q)$  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)

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

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

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

□

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  $a_1|a_2|\dots|a_k$  is a bunch consisting of the values of  $a_1, \dots, a_k$ . 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  $P \triangleright a$  is a guarded expression: if the guard fails, then the entire expression fails; otherwise the value of  $P \triangleright a$  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  $S \rightarrow \alpha \cdot$ .

$$\begin{aligned}
[q](\xi, c_1, \dots, c_{\text{active}(q)}) &:= \\
\text{letrec } c_0(X, \xi) &= \\
&\quad [\text{goto}(q, X)](\xi, c_0, c_1, \dots, c_{\text{active}(\text{goto}(q, X)) - 1}) \\
\text{in } A \rightarrow \alpha \cdot (\rho) \in \bar{q} \wedge \xi|_k = \rho &\triangleright c_{|\alpha|}(A, \xi) \\
| \xi = x\xi' \wedge x \in \text{nextterm}(q) &\triangleright c_0(x, \xi')
\end{aligned}$$
Figure 2.3: Functional LR( $k$ ) parser, continuation-based version

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

$$[q_f](\epsilon, c_1, \dots, c_{\text{active}(q)}) := \text{succeed}$$

### 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( $k$ ).

### 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  $n \rightarrow \nu(\tau)$  where  $\tau \in \text{first}_k(\text{la\_suffix})$ :

```

  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` function 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 lookahead 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 automata [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  $q_0^{(0)}, \dots, q_n^{(0)}$  has states closures  $q_0, \dots, q_n$ . In contrast to the LR(k) parser, the SLR(k) automaton has the following states:

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

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

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

The state transition goto is also just a variant the LR(0) case here called goto<sup>(0)</sup>:

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

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  $\text{follow}_k$  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 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:

$$\begin{aligned} \langle \text{exp} \rangle &::= \langle \text{term} \rangle \mid \mathbf{error} \mid \langle \text{term} \rangle + \langle \text{exp} \rangle \mid \langle \text{term} \rangle - \langle \text{exp} \rangle \\ \langle \text{term} \rangle &::= \langle \text{prod} \rangle \mid \langle \text{prod} \rangle * \langle \text{term} \rangle \mid \langle \text{prod} \rangle / \langle \text{term} \rangle \\ \langle \text{prod} \rangle &::= \mathbf{number} \mid ( \langle \text{exp} \rangle ) \mid ( \mathbf{error} ) \end{aligned}$$

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.

$$\begin{array}{ll}
 \langle \text{exp} \rangle ::= \langle \text{term} \rangle & \langle \text{exp} \rangle.v = \langle \text{term} \rangle.v \\
 \quad | \langle \text{term} \rangle + \langle \text{exp} \rangle & \langle \text{exp}_1 \rangle.v = \langle \text{term} \rangle.v + \langle \text{exp}_2 \rangle.v \\
 \quad | \langle \text{term} \rangle - \langle \text{exp} \rangle & \langle \text{exp}_1 \rangle.v = \langle \text{term} \rangle.v - \langle \text{exp}_2 \rangle.v \\
 \langle \text{term} \rangle ::= \langle \text{prod} \rangle & \langle \text{term} \rangle.v = \langle \text{prod} \rangle.v \\
 \quad | \langle \text{prod} \rangle * \langle \text{term} \rangle & \langle \text{term}_1 \rangle.v = \langle \text{prod} \rangle.v \cdot \langle \text{term}_2 \rangle.v \\
 \quad | \langle \text{prod} \rangle / \langle \text{term} \rangle & \langle \text{term}_1 \rangle.v = \langle \text{prod} \rangle.v / \langle \text{term}_2 \rangle.v \\
 \langle \text{prod} \rangle ::= 42 & \langle \text{prod} \rangle.v = 42 \\
 \quad | ( \langle \text{exp} \rangle ) & \langle \text{prod} \rangle.v = \langle \text{exp} \rangle.v
 \end{array}$$

$$\begin{array}{ll}
 \langle \text{bit} \rangle ::= 0 & \langle \text{bit} \rangle.v = 0 \\
 \quad | 1 & \langle \text{bit} \rangle.v = 2^{\langle \text{bit} \rangle.s} \\
 \langle \text{bits} \rangle ::= \langle \text{bit} \rangle & \langle \text{bits} \rangle.v = \langle \text{bit} \rangle.v, \langle \text{bit} \rangle.s = \langle \text{bits} \rangle.s, \langle \text{bits} \rangle.l = 1 \\
 \quad | \langle \text{bits} \rangle \langle \text{bit} \rangle & \langle \text{bits}_1 \rangle.v = \langle \text{bits}_2 \rangle.v + \langle \text{bit} \rangle.v, \langle \text{bit} \rangle.s = \langle \text{bits}_1 \rangle.s, \\
 & \langle \text{bits}_2 \rangle.s = \langle \text{bits}_1 \rangle.s + 1, \langle \text{bits}_1 \rangle.l = \langle \text{bits}_2 \rangle.l + 1 \\
 \langle \text{num} \rangle ::= \langle \text{bits} \rangle & \langle \text{num} \rangle.v = \langle \text{bits} \rangle.v, \langle \text{bits} \rangle.s = 0 \\
 \quad | \langle \text{bits} \rangle . \langle \text{bits} \rangle & \langle \text{num} \rangle.v = \langle \text{bits}_1 \rangle.v + \langle \text{bits}_2 \rangle.v, \langle \text{bits}_1 \rangle.s = 0, \\
 & \langle \text{bits}_2 \rangle.s = -\langle \text{bits}_2 \rangle.l
 \end{array}$$

### 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,  $\langle \circ A \rightarrow \alpha \rangle$  identifies the left-hand side  $A$ , whereas  $\langle A \rightarrow \alpha \circ X \beta \rangle$  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  $A$ .  $\text{Syn}(A) \cap \text{Inh}(A) = \emptyset$  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  $\langle \circ A \rightarrow \alpha \rangle.a$  with  $a \in \text{Att}(A)$  or  $\langle A \rightarrow \gamma \circ B \delta \rangle.a$  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  $\text{Def}(A \rightarrow \alpha)$  and the applied occurrences  $\text{App}(A \rightarrow \alpha)$ .

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

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

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

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

Actually, the attribute dependencies of the grammar according to the above definition 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

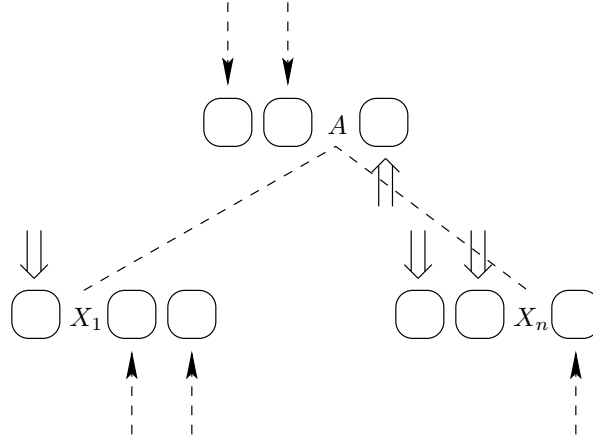


Figure 2.4: A grammar attribution

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  $a \in \text{Att}(A)$ . For a node  $n$ , let  $A \rightarrow \alpha$  be the associated production. Furthermore, let, for a position  $p = \langle A \rightarrow \gamma \circ B \delta \rangle$  with  $\gamma B \delta = \alpha$ ,  $n^p$  be the corresponding child node, and, for a position  $p = \langle \circ A \rightarrow \alpha \rangle$ ,  $n^p = n$ . Then, for all  $d \in \text{Def}(A \rightarrow \alpha)$  with  $d = p.a$ , the following must hold:

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

where  $a_j = p_j.a'_j$  for all  $j$  for rule  $d = f_d(a_1, \dots, a_{i_d})$ .  $\square$

Hence, an attributed grammar  $G$  with start production  $S \rightarrow A$  defines a meaning function  $\mathcal{M} : T^* \times D^{|\text{Inh}(A)|} \rightarrow D^{|\text{Syn}(S)|}$ . For  $\xi \in L(G)$ ,  $\mathcal{M}(\xi, v_1, \dots, v_{|\text{Inh}(A)|})$  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

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

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

In an L-attributed grammar, any attribute occurrence may only depend on occurrences 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  $\text{Inh}(A) = \emptyset$  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  $\text{Syn}(A) = \{v\}$  for all  $A \in N$ . Also, for simplicity of the presentation,  $f_{\langle A \rightarrow \alpha \rangle . v}$  always has  $|\alpha|$  arguments. Terminals simply yield some reserved value  $\perp$  as their attribute which may not be

used.

$$\begin{aligned}
[q](\xi, c_1, \dots, c_{\text{nactive}(q)}, v_1, \dots, v_{\text{nactive}(q)}) &:= \\
\text{letrec } c_0(X, v_0, \xi) &= \\
& \quad [\text{goto}(q, X)](\xi, c_0, c_1, \dots, c_{\text{nactive}(\text{goto}(q, X))-1}, v_0, v_1, \dots, v_{\text{nactive}(\text{goto}(q, X))-1}) \\
\text{in } A \rightarrow \alpha \cdot (\rho) \in \bar{q} \wedge \xi|_k = \rho & \triangleright c_{|\alpha|}(A, f_{\langle \circ A \rightarrow \alpha \rangle.v}(v_{|\alpha|}, \dots, v_1), \xi) \\
| \xi = x\xi' \wedge x \in \text{nextterm}(q) & \triangleright c_0(x, \perp, \xi')
\end{aligned}$$

For implementation purposes, it now becomes clear what the **attribution** component of a grammar rule is—it is that function  $f_{\langle A \rightarrow \alpha \rangle.v}$ , 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 is not a sequence of terminals  $\xi$  but rather a sequence of terminal(“token”)/attribute pairs  $\omega$ . In the following specification,  $\pi_1^* : (T \times D)^* \rightarrow T^*$  just extracts the terminals from a sequence of terminal/attribute pairs.

$$\begin{aligned}
[q](\omega, c_1, \dots, c_{\text{nactive}(q)}, v_1, \dots, v_{\text{nactive}(q)}) &:= \\
\text{letrec } c_0(X, v_0, \omega) &= \\
& \quad [\text{goto}(q, X)](\omega, c_0, c_1, \dots, c_{\text{nactive}(\text{goto}(q, X))-1}, v_0, v_1, \dots, v_{\text{nactive}(\text{goto}(q, X))-1}) \\
\text{in } A \rightarrow \alpha \cdot (\rho) \in \bar{q} \wedge (\pi_1^*(\omega))|_k = \rho & \triangleright c_{|\alpha|}(A, f_{\langle A \rightarrow \alpha \rangle.v}(v_{|\alpha|}, \dots, v_1), \omega) \\
| \omega = (x, v_0)\omega' \wedge x \in \text{nextterm}(q) & \triangleright c_0(x, v_0, \omega')
\end{aligned}$$

## 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

```

⟨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

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  $\Sigma$  be the alphabet of the language of a program. Abstract syntax generation is a function

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

where  $D$  is a suitable set such that a function

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

exists with

$$parse \circ unparse = id_D.$$

□

```

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

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.

# 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.
- [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.
- [FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.
- [FHP92] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. *Letters on Programming Languages and Systems*, 1(3):213–226, 1992.
- [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  $\lambda$ -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, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 146–155, La Jolla, CA, 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.