Systematic Compiler Construction

Michael Sperber Peter Thiemann

January 26, 2022

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, 2004, 2006, 2022

Contents

Chapter 2

Syntax 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 grammars for this purpose, although programming languages are not context-free, in general (for example, most languages require that variables be defined before use). The reasons for choosing context-free grammars here are that they are reasonably expressive and concise and that there are large subclasses for which the word problem is decidable in linear time.

```
⟨ObjectLiteral⟩ ::=
    { }
    | { ⟨PropertyNameAndValueList⟩ }
⟨PropertyNameAndValueList⟩ ::=
    ⟨PropertyName⟩ : ⟨AssignmentExpression⟩
    | ⟨PropertyNameAndValueList⟩ , ⟨PropertyName⟩ : ⟨AssignmentExpression⟩
⟨PropertyName⟩ ::=
    ⟨Identifier⟩
    | ⟨StringLiteral⟩
    | ⟨NumericLiteral⟩
⟨AssignmentExpression⟩ ::=
    ...⟨ObjectLiteral⟩
```

Figure 2.1: Partial grammar for JavaScript object literals.

Figure 2.1 shows a context-free grammar for JavaScript object literals. The language defined by the grammar is obviously not regular: it contains recursion because the ⟨ObjectLiteral⟩ nonterminal is derivable from the ⟨AssignmentExpression⟩. 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 `->`.

Syntax 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.

- Construct a derivation tree for the program.

Parsing comes essentially in two flavors.

- A *top-down* or *recursive descent* parser constructs a derivation tree “depth-first” by starting from the root of the tree.
- A *bottom-up* or *recursive ascent* parser constructs a derivation tree starting from the leftmost leaves of the tree.

After establishing some notation, the examination of both parsing technologies follow the same pattern. Starting from a non-deterministic specification, we discover sufficient conditions for making this specification deterministic.

2.1 Context-Free Grammars

2.1.1 Definitions

We need some notation for dealing with words and context-free grammars.

2.1 Definition (Sequence Manipulation)
Let M be an alphabet and M^* the set of sequences (words) over M.

- $\epsilon \in M^*$ is the empty sequence.
- The length $|w| = n$ if $w = x_1 \dots x_n \in M^*$ with $x_i \in M$.
- w^k denotes a sequence with k copies of w.
- $w_{|k} = x_1 \dots x_{\min(k,n)}$ if $w = x_1 \dots x_n$; that is, the k-prefix of w.

□

2.2 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* with $N \cap T = \emptyset$, $S \in N$ 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 $\alpha \in V^*$ of grammar symbols.

□

Some letters denote elements of certain sets by default:

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

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

2.3 Definition (Derives Relation)
G induces the *derives relation* $\Rightarrow \subseteq V^* \times V^*$ 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 $\alpha_0, \alpha_1, \dots, \alpha_n$ where $\alpha_{i-1} \Rightarrow \alpha_i$ for $1 \le i \le n$. A *sentential form* is a sequence appearing in a derivation beginning with the start symbol.

□

The language defined by a context-free grammar is the set of terminal strings derivable from the start symbol.

$$L(G) = \{w \mid S \overset{*}{\Rightarrow} w\}$$

2.4 Definition (Leftmost Derivation)
The *leftmost derivation relation* $\Rightarrow^l \subseteq V^* \times V^*$ is the least relation such that

$$wA\beta \Rightarrow^l w\alpha\beta \;:\Leftrightarrow\; A \to \alpha$$

The terms *leftmost derivation* and *left sentential form* as well as the reflexive transitive closure are defined analogously as for a derivation.

□

2.1.2 Representation

Context-free grammars are easily represented in Python.

```
TS    = Any
NTS   = Any

@dataclass(frozen=True)
class NT:
    nt: Any

Symbol = NT | TS

@dataclass(frozen=True)
class Production:
    lhs: NTS
    rhs: list[Symbol]
    ext: Any = None

@dataclass(frozen=True)
class Grammar:
    nonterminals: list[NTS]
    terminals: list[TS]
    rules: list[Production]
    start: NTS

    def productions_with_lhs(self, nts: NTS) -> list[Production]:
        return [ rule for rule in self.rules if rule.lhs == nts ]
```

Here is an example grammar for some arithmetic expressions:

3 Example
Consider the following grammar for expressions.

$$\begin{array}{lcl} T & \to & E \mid T\texttt{+}E \\ E & \to & F \mid E\texttt{*}F \\ F & \to & \texttt{x} \mid \texttt{2} \mid \texttt{(}T\texttt{)} \end{array}$$

□

```
expr_grammar = Grammar(
    ['T', 'E', 'F'],
    ['x', '2', '(', ')', '+', '*'],
    [Production('T', [NT('E')])
    ,Production('T', [NT('T'), '+', NT('E')])
    ,Production('E', [NT('F')])
    ,Production('E', [NT('E'), '*', NT('F')])
    ,Production('F', ['x'])
    ,Production('F', ['2'])
    ,Production('F', ['(', NT('T'), ')'])],
    'T'
)
```

2.2 Recursive-Descent Parsing

The introduction of parsing requires sufficient formal background that it is easier to 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. We first generalize this notion slightly by having parsers recognize and chop away prefixes that they recognize. This generalization makes parsers composable and allows us to derive a recursive-descent parser from a given grammar.

2.5 Definition
A (non-deterministic) parser for a language L is a function

$$p_L : T^* \to \mathcal{P}\, T^*$$

such that $p_L(w) = \{v \in T^* \mid (\exists u \in L)\ w = uv\}$.

□

With this definition it holds that $v \in p_L(uv)$ iff $u \in L$. As a special case we find that $w \in L \Leftrightarrow \epsilon \in p_L(w)$.

Let's now specialize L to the language $L(\alpha) = \{w \in T^* \mid \alpha \overset{*}{\Rightarrow} w\}$ recognized from a sequence of grammar symbols α. We call this parser $[\alpha] : T^* \to \mathcal{P}(T^*)$ and—by definition—it satisfies the following equation:

$$[\alpha](w) = \{v \in T^* \mid \alpha \overset{*}{\Rightarrow} u, w = uv\}$$

Hence, a sequence of terminals w is in the language defined by the grammar iff $\epsilon \in [S](w)$. Consequently, the above equation specifies a *recognizer* for the grammar. The specification leads directly to an implementation by specializing with respect to α. This program derivation technique is called *constructive induction.*

If $\alpha = \epsilon$, then the parser $[\epsilon]$ can be calculated as follows.

$$\begin{aligned}
[\epsilon](w) &= \{v \mid \epsilon \overset{*}{\Rightarrow} u, w = uv\} \\
&= \{v \mid w = v\} \\
&= \{w\}
\end{aligned}$$

If $\alpha = X\beta$ an analogous calculation yields

$$\begin{aligned}
[X\beta](w) &= \{v \mid X\beta \overset{*}{\Rightarrow} u, w = uv\} \\
&= \{v \mid X \overset{*}{\Rightarrow} u_1, w = u_1 w', \beta \overset{*}{\Rightarrow} u_2, w' = u_2 v\} \\
&= \{v \mid w' \in [X](w), v \in [\beta](w')\} \\
&= \bigcup\{[\beta](w') \mid w' \in [X](w)\}
\end{aligned}$$

That is, the parser first computes the rest w' after consuming the X part of the input w and then applies the parser $[\beta]$ to that rest w'.

It remains to define the primitive parsers for terminal and nonterminal symbols. For terminals, the solution is immediate:

$$[x](x_1 \dots x_n) = \begin{cases} \{x_2 \dots x_n\} & \text{if } n > 1 \text{ and } x_1 = x \\ \varnothing & \text{otherwise} \end{cases}$$

For a nonterminal, we take the union of the right hand sides of all productions for that nonterminal.

$$\begin{array}{rcl} [A](w) & = & \{v \mid A \overset{*}{\Rightarrow} u, w = uv\} \\ & = & \{v \mid A \Rightarrow \alpha \overset{*}{\Rightarrow} u, A \to \alpha \in P, w = uv\} \\ & = & \bigcup_{A\to\alpha}[\alpha](w) \end{array}$$

The last part of the definition recursively descends into the right-hand sides of the grammar rules of a nonterminal. For that reason, this kind of parser is called a *recursive-descent parser*. For each A there is a definition of $[A]$, which only depends on the definitions of the other parsers $[A_i]$ (recursively) and on the $[x_i]$.

4 Example
Consider the following grammar for expressions.

$$\begin{array}{rcl} T & \to & E \mid T\texttt{+}E \\ E & \to & F \mid E\texttt{*}F \\ F & \to & \texttt{x} \mid \texttt{2} \mid \texttt{(}T\texttt{)} \end{array}$$

It gives rise to the following nondeterministic parser:

$$\begin{array}{rcl} [T](w) & = & [E](w) \cup [T\texttt{+}E](w) \\ & = & [E](w) \cup \bigcup\{[\texttt{+}E](u) \mid u \in [T](w)\} \\ [E](w) & = & [F](w) \cup [E\texttt{*}F](w) \\ [F](w) & = & [\texttt{x}](w) \cup [\texttt{2}](w) \cup [\texttt{(}T\texttt{)}](w) \end{array}$$

□

It is a straightforward to translate the above definition into Prolog, Haskell, or even into Python. In particular, we can represent the non-determinism using generators and backtracking.

```
def td_parse(g: Grammar, alpha: list[Symbol], inp: list[TS]
          ) -> Iterator[list[TS]]:
    match alpha:
        case []:
            yield inp
        case [NT(nt), *rest_alpha]:
            for rule in g.productions_with_lhs(nt):
                for rest_inp in td_parse(g, rule.rhs, inp):
                    yield from td_parse(g, rest_alpha, rest_inp)
        case [ts, *rest_alpha]:
            if inp and ts == inp[0]:
                yield from td_parse(g, restrict, inp[1:])
```

The resulting parser has two significant problems, which renders it hardly usable:

1. The specified parser is *non-deterministic*: $[A](w)$ dives down into the right-hand sides of all grammar productions of A. This behavior might cause the implementation to run in time exponential in the length of the input string!

2. Consider the parser for the expression grammar from Example 4 and in particular the function for the nonterminal T.

$$\underline{[T](w)} = [E](w) \cup \bigcup \{[+E](u) \mid u \in \underline{[T](w)}\}$$

Evidently, $[T]$ is called recursively with the same argument so that the definition leads to infinite recursion. In general, a recursive descent parser for a grammar which contains so-called *left-recursive* rules—rules which have their left-hand-sides also as the first symbol of their right-hand sides—exhibits this behavior.

Both problems are fixable and the next two subsections give details on how to proceed if a grammar exhibits these problems.

Left Recursion

The expression grammar from Example 4 results in a recursive-descent parser that never terminates. For example, the parser $[T]$ of the expression grammar immediately calls $[T]$ again on the same input. To fix this problem, we have to recognize grammars exhibiting such undesirable recursive behavior and transform them using *right factorization.*

2.6 Definition
A context-free grammar G is *left-recursive* if it contains a left-recursive nonterminal. Nonterminal A is *left-recursive* if there is some $\alpha \in V^*$ such that $A \overset{+}{\Rightarrow} A\alpha$. A is *directly left-recursive* if there is a production $A \to A\alpha$.

□

Left recursion is the culprit for the bad behavior of the recursive descent parser. It must be eliminated because recursive descent parsers do not work for left-recursive grammars. In the following we assume that the given grammar has no cycles (derivations of the form $A \overset{+}{\Rightarrow} A$) and is ε-free (it has no ε-productions of the form $A \to \varepsilon$). These assumptions do not restrict the grammars we can work with because there are algorithms to eliminate cycles and ε-productions.

The general elimination procedure calls on the elimination of direct left-recursion, so we consider it first. To eliminate direct left-recursion for nonterminal A we collect all rules for A. Let

$$A \to \alpha_1 \mid \cdots \mid \alpha_m \mid A\beta_1 \mid \cdots \mid A\beta_n$$

be all those rules where none of the α_i starts with A. Introduce a new nonterminal A' and replace the productions by

$$\begin{array}{lcl} A & \to & \alpha_1 A' \mid \cdots \mid \alpha_m A' \\ A' & \to & \epsilon \mid \beta_1 A' \mid \ldots \beta_n A' \end{array}$$

This transformation does not change the language of the grammar, but eliminates all occurrences of direct left-recursion. (It must be that $\alpha_i \neq \varepsilon$ and $\beta_j \neq \varepsilon$ for all i, j because the grammar contains neither cycles nor ε-productions.) It must be applied to all directly left-recursive nonterminals.

Indirect left-recursion is transformed in direct left-recursion to be eliminated by the transformation just described. The idea of the transformation is to consider the nonterminals in an arbitrary, fixed order $(A_i \mid i \in \{1, \ldots, n\})$. For each i, the first step removes all occurrences of rules of the form $A_i \to A_j\alpha$ where $j < i$ by substituting all right-hand-sides of A_j (this step also requires the grammar to be ε-free). The second step eliminates direct left-recursion for A_i. After these two steps, no symbols A_j for $j \leq i$ occur at the left end of a producation for A_k $(k \leq i)$. Thus, after processing A_n, the grammar does not contain left recursion anymore.

5 Example
The grammar for expressions exhibits direct left recursion in the symbols T and E.

$$\begin{array}{lcl} T & \to & E \mid T\texttt{+}E \\ E & \to & F \mid E\texttt{*}F \\ F & \to & \texttt{x} \mid \texttt{2} \mid \texttt{(}T\texttt{)} \end{array}$$

Applying left recursion elimination to T and E yields the grammar

$$\begin{array}{lcl} T & \to & ET' \\ T' & \to & \varepsilon \mid \texttt{+}ET' \\ E & \to & FE' \\ E' & \to & \varepsilon \mid \texttt{*}FE' \\ F & \to & \texttt{x} \mid \texttt{2} \mid \texttt{(}T\texttt{)} \end{array}$$

□

Lookahead

Fortunately, it is possible to fix the exponential blow-up problem, too. The idea is to change the definition of $[A]$ so that it does not union over several right-hand sides but instead picks just one of them immediately. The resulting parsing algorithm is linear in the size of the input. 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.

2.7 Definition (first_k, follow_k)
For an integer $k \geq 0$, first_k and follow_k are defined as follows:

$$\begin{array}{rcl} \mathrm{first}_k & : & V^* \to \mathcal{P}(T^{\leq k}) \\ \mathrm{first}_k(\alpha) & := & \left\{ w_{|k} \mid \alpha \overset{*}{\Rightarrow} w \right\} \\ \mathrm{follow}_k & : & N \to \mathcal{P}(T^k) \\ \mathrm{follow}_k(A) & := & \{ w \mid S \overset{*}{\Rightarrow} \beta A \gamma \ \wedge \ w \in \mathrm{first}_k(\gamma) \} \end{array}$$

□

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. Its companion function, $\mathrm{follow}_k(A)$, computes the set of all terminal sequences of length k that may follow A in a sentential form.

These two functions are put to use as follows. Consider a leftmost derivation

$$S \overset{*}{\Rightarrow} vA\beta \Rightarrow v\alpha\beta \overset{*}{\Rightarrow} vw$$

in which the parser $[A]$ has to make the decision for the production $A \to \alpha$ when seeing w, or rather the first k symbols thereof. The k-prefix of w has the following property

$$w_{|k} \in \mathrm{first}_k(\alpha\beta) = (\mathrm{first}_k(\alpha)\,\mathrm{first}_k(\beta))_{|k} \subseteq (\mathrm{first}_k(\alpha)\,\mathrm{follow}_k(A))_{|k}$$

This observation leads to the definition of the lookahead set of a production and the notion of a strong LL(k) grammar. (The first “L” is for “parsable from left to right,” the second “L” for “generate a leftmost derivation”.)

2.8 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 a *strong LL(k) grammar* if, for productions $A \to \alpha$ and $A \to \beta$ with $\alpha \neq \beta$,

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

□

For a strong LL(k) grammar, the nondeterministic choice in the parser for nonterminals becomes a conditional:

$$[A](w) = \begin{cases} [\alpha_1](w) & \text{if } w_{|k} \in \mathrm{LLLA}_k(A \to \alpha_1) \\ \vdots & \\ [\alpha_n](w) & \text{if } w_{|k} \in \mathrm{LLLA}_k(A \to \alpha_n) \\ \emptyset & \text{otherwise} \end{cases}$$

The traditional notion of an LL(k) grammar encompasses slightly more grammars, but is less easy to test for $k > 1$.

2.9 Definition
A context-free grammar is an *LL(k) grammar* if, for all derivations of the form

$$\begin{array}{lllllll} S & \overset{*}{\Rightarrow}^l & wA\alpha & \Rightarrow^l & w\beta\alpha & \overset{*}{\Rightarrow}^l & wu \\ S & \overset{*}{\Rightarrow}^l & wA\alpha & \Rightarrow^l & w\gamma\alpha & \overset{*}{\Rightarrow}^l & wv \end{array}$$

$u_{|k} = v_{|k}$ implies $\beta = \gamma$.

□

That is, whenever there is a choice of productions in a leftmost derivation, the first k symbols of the remaining input determine which production must be used. This definition is a bit unwieldy, but there is an equivalent formulation [WM96].

2.10 Theorem
G is LL(k) if and only if for all productions $A \to \alpha$, $A \to \beta$ with $\alpha \neq \beta$, and for all leftmost derivations $S \overset{*}{\Rightarrow}^l wA\gamma$, it holds that

$$\mathrm{first}_k(\alpha\gamma) \cap \mathrm{first}_k(\beta\gamma) = \emptyset$$

It turns out that each strong LL(1) grammar is also an LL(1) grammar (see [WM96]), but for $k > 1$ there are LL(k) grammars which are not strong LL(k).

6 Example
Let G have productions $S \to aAaa \mid bAba$ and $A \to b \mid \epsilon$. G is LL(2) because

- for $S \Rightarrow aAaa$ we have disjoint lookahead sets: $\mathrm{first}_2(baa) = \{ba\}$ and $\mathrm{first}_2(aa) = \{aa\}$
- for $S \Rightarrow bAaa$ we have disjoint lookahead sets, too: $\mathrm{first}_2(bba) = \{bb\}$ and $\mathrm{first}_2(ba) = \{ba\}$.

However, the strong LL(2) condition is violated because $\mathrm{follow}_2(A) = \{aa, ba\}$ so that the lookahead sets for the A productions are not disjoint:

$$\begin{array}{ll} LLLA_2(A \to b) & = \mathrm{first}_2(b\,\mathrm{follow}_2(A)) = \{ba, bb\} \\ LLLA_2(A \to \epsilon) & = \mathrm{first}_2(\epsilon\,\mathrm{follow}_2(A)) = \{aa, ba\} \end{array}$$

□

7 Example
We recall the expression grammar from Example 5 after elimination of left recursion:

$$\begin{array}{llll} T & \to ET' & T' & \to \epsilon \mid \texttt{+}ET' \\ E & \to FE' & E' & \to \epsilon \mid \texttt{*}FE' \\ F & \to \texttt{x} \mid \texttt{2} \mid \texttt{(}T\texttt{)} & & \end{array}$$

This grammar is LL(1) because the lookahead sets of each pair of productions for T', E', and F are disjoint. For T and E there is only one production, so their lookahead can be ignored.

$$\begin{array}{lcl} \mathrm{LLLA}_1(T' \to \epsilon) & = & \{\epsilon, \texttt{)}\} \\ \mathrm{LLLA}_1(T' \to \texttt{+}ET') & = & \{\texttt{+}\} \\ \mathrm{LLLA}_1(E' \to \epsilon) & = & \{\epsilon, \texttt{)}, \texttt{+}\} \\ \mathrm{LLLA}_1(E' \to \texttt{*}FE') & = & \{\texttt{*}\} \\ \mathrm{LLLA}_1(F \to \texttt{x}) & = & \{\texttt{x}\} \\ \mathrm{LLLA}_1(F \to \texttt{2}) & = & \{\texttt{2}\} \\ \mathrm{LLLA}_1(F \to \texttt{(}T\texttt{)}) & = & \{\texttt{(}\} \end{array}$$

2.2.2 Implementing Recursive-Descent Parsing

With the formal specification at hand, a module 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`.

TO BE COMPLETED

2.2.3 Computation of First Sets

As an example for computing the lookahead functions first_k and follow_k, we derive an implementation of the first_k function. The first step derives an alternative definition of first_k as a recursive system of equations. We obtain this definition again by applying the principle of constructive induction to the definition of first_k, *i.e.*, $\mathrm{first}_k(\alpha) = \{w_{|k} \mid \alpha \overset{*}{\Rightarrow} w\}$.

First, we decompose the sequence α, which may be empty or nonempty. For $\alpha = \varepsilon$, we obtain

$$\begin{array}{lcl} \mathrm{first}_k(\varepsilon) & = & \{w_{|k} \mid \varepsilon \overset{*}{\Rightarrow} w\} \\ & = & \{\varepsilon\} \end{array}$$

For $\alpha = X\beta$, we obtain

$$\begin{array}{lcl} \mathrm{first}_k(X\beta) & = & \{w_{|k} \mid X\beta \overset{*}{\Rightarrow} w\} \\ & = & \{(vu)_{|k} \mid X \overset{*}{\Rightarrow} v, \beta \overset{*}{\Rightarrow} u\} \\ & = & (\mathrm{first}_k(X)\,\mathrm{first}_k(\beta))_{|k} \end{array}$$

Finally, a case distinction on X. The case $X = x$ is trivial:

$$\mathrm{first}_k(x) = \{x\}$$

The case $X = A$:

$$\begin{array}{lcl} \mathrm{first}_k(A) & = & \{w_{|k} \mid A \overset{*}{\Rightarrow} w\} \\ & = & \{w_{|k} \mid A \Rightarrow \alpha \overset{*}{\Rightarrow} w, A \to \alpha\} \\ & = & \bigcup_{A \to \alpha} \mathrm{first}_k(\alpha) \end{array}$$

Essentially, this transformation yields a specification of first_k as a system of equations with unknowns $\mathrm{first}_k(A)$, for $A \in N$. The solution for such a system is a vector $\overline{L} = (L_A \subseteq T^{\leq k} \mid A \in N) \in \prod_{A \in N} \mathcal{P}(T^{\leq k})$ of sets of terminal strings of length $\leq k$. To solve such an equation, we consider the right-hand sides of the definitions as functions $\mathcal{F}_k : \prod_{A \in N} \mathcal{P}(T^{\leq k}) \to V^* \to \mathcal{P}(T^{\leq k})$ that improve an approximation to the solution.

$$\begin{aligned}
\mathcal{F}_k(\overline{L})(\varepsilon) &= \{\varepsilon\} \\
\mathcal{F}_k(\overline{L})(X\beta) &= (\mathcal{F}_k(\overline{L})(X)\mathcal{F}_k(\overline{L})(\beta))_{|k} \\
\mathcal{F}_k(\overline{L})(x) &= \{x\} \\
\mathcal{F}_k(\overline{L})(A_i) &= L_{A_i}
\end{aligned}$$

Given a vector $\overline{L} \in \prod_{A \in N} \mathcal{P}(T^{\leq k})$ of approximations, an improved approximation for $A \in N$ as computed as follows:

$$L'_A = \bigcup_{A \to \alpha \in P} \mathcal{F}_k(\overline{L})(\alpha)$$

In the following, $\overline{\mathcal{F}_k}(\overline{L})$ stands for the vector $(\bigcup_{A \to \alpha \in P} \mathcal{F}_k(\overline{L})(\alpha) \mid A \in N)$. Here are some facts about the functions $\mathcal{F}_k$:

1. If $\overline{L}$ is a solution, then $\overline{L} = \overline{\mathcal{F}_k}(\overline{L})$.
2. If $\overline{M} \subseteq \overline{L}$ then $\overline{\mathcal{F}_k}(\overline{M}) \subseteq \overline{\mathcal{F}_k}(\overline{L})$ (monotony).
3. Let $\overline{L}$ be a solution. If $\overline{M} \subseteq \overline{L}$ then $\overline{\mathcal{F}_k}(\overline{M}) \subseteq \overline{L}$.
4. $\overline{L} = \bigcup_{i \in \mathbf{N}} \overline{\mathcal{F}_k^{(i)}}(\overline{\emptyset})$ is a solution.

Hence, the solution (*i.e.*, the first_k function) can be computed by *fixpoint iteration*, that is

- start with $\overline{L}_0 = \overline{\emptyset}$,
- apply $\overline{L}_{i+1} = \overline{\mathcal{F}_k}(\overline{L}_i)$ until $\overline{L}_{i+1} = \overline{L}_i$.

This iteration terminates because $\overline{\mathcal{F}_k}$ is monotone and because $T^{\leq k}$ is finite.

To compute the follow function, a similar strategy applies by exploiting the following lemma.

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

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

□

2.2.4 Computation of First(1) Sets

As a special case, we consider the computation of first_1 sets. As a first step, we need to find out, whether a nonterminal of a grammar can derive the empty string ε. This information can be calculated with a fixed-point computation that produces a mapping from nonterminals to booleans. A nonterminal A is mapped to `True` as soon as we can prove that $A \overset{*}{\Rightarrow} \varepsilon$.

```
EmptySet = dict[NTS,bool]
```

Initially, nothing is known about each nonterminal:

```
def initial_empty(g: Grammar) -> EmptySet:
    return { n : False for n in g.nonterminals }
```

Given an approximative `es: EmptySet`, we can evaluate whether a symbol string `alpha` may derive ε as follows:

```
def derives_empty(es: EmptySet, alpha: list[Symbol]) -> bool:
    match alpha:
        case []:
            return True
        case [NT(nt), *rest]:
            return es[nt] and derives_empty(es, rest)
        case [ts, *rest]:
            return False
```

We can use this function to update out knowledge about each nonterminal by evaluating its right-hand side:

```
def update_empty(g: Grammar, fs: EmptySet):
    for n in g.nonterminals:
        fn = fs[n]
        for rule in g.productions_with_lhs(n):
            fn = fn or derives_empty(fs, rule.rhs)
        fs[n] = fn
```

It remains to compute the fixed point of this function. We provide a general function to do so because we can reuse it to compute the actual first sets.

```
def fixed_point(current_map: dict, update: Callable[[FirstSet], None]) -> dict:
    next_map = None
    while next_map is None or \
            any(current_map[k] != next_map[k] for k in current_map):
        next_map = current_map
        current_map = current_map.copy()
        update(current_map)
    return current_map
```

The calculation of the emptiness information is a straightforward application of the `fixed_point` function.

```
def calculate_empty(g: Grammar) -> EmptySet:
    es = initial_empty(g)
    return fixed_point(es, partial(update_empty, g))
```

2.3 Bottom-Up Parsing

Recursive-descent parsing is simple to implement, but requires an LL(k) grammar to be effective. While 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 known as *bottom-up* or *LR* parsing, where LR stands for **l**eft-to-right processing of the input and construction of a reversed **r**ightmost derivation.) 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 Overview of Bottom-Up Parsing

A bottom-up parser constructs a reversed rightmost derivation while processing the input. Intuitively, it starts building the derivation tree from the leftmost corner by accumulating a right-sentential form.

The typical state of a bottom-up parser is a pair $\alpha \bullet w$ of a stack α and the remaining input w, so that αw is a right-sentential form. Parsing proceeds according to the following steps.

1. The initial state is $\bullet w$, that is, the stack is empty and the full input is available.

2. In state $\alpha \bullet w$, apply one of the following alternatives

   (a) If $\alpha = \beta\gamma$ such that $A \to \gamma \in P$, then **reduce** this production and change state to $\beta A \bullet w$.

   (b) If $w = \mathtt{a}w'$, then **shift** the terminal `a` and change state to $\alpha\mathtt{a} \bullet w'$.

   (c) If $\alpha = S$ and $w = \epsilon$, then parsing finishes with success.

   Reject the input if none of the alternatives applies.

3. If the stack of the current state is such that a reduction state is eventually reachable, then continue with item 2. Otherwise reject the input.

Item 2 is nondeterministic in several respects. There may be more than one way to split α into β and the *handle* γ; for a chosen handle γ there may be several rules with right side γ; the parser could shift instead of trying to reduce.

Evidently, the work horses of the parser are the two actions **reduce** and **shift**. Hence, bottom-up parsers are often called shift-reduce parsers.

8 Example
Let's trace a shift-reduce parser accepting the word `2+x*x` using the grammar for

arithmetic expressions from Example 5.

$\bullet\texttt{2+x*x}$	shift
$\texttt{2}\bullet\texttt{+x*x}$	reduce $F \to \texttt{2}$
$F\bullet\texttt{+x*x}$	reduce $E \to F$
$E\bullet\texttt{+x*x}$	reduce $T \to E$
$T\bullet\texttt{+x*x}$	shift
$T\texttt{+}\bullet\texttt{x*x}$	shift
$T\texttt{+x}\bullet\texttt{*x}$	reduce $F \to \texttt{x}$
$T\texttt{+}F\bullet\texttt{*x}$	reduce $F \to E$
$T\texttt{+}E\bullet\texttt{*x}$	shift
$T\texttt{+}E\texttt{*}\bullet\texttt{x}$	shift
$T\texttt{+}E\texttt{*x}\bullet$	reduce $F \to \texttt{x}$
$T\texttt{+}E\texttt{*}F\bullet$	reduce $E \to E\texttt{*}F$
$T\texttt{+}E\bullet$	reduce $T \to T\texttt{+}E$
$T\bullet$	success

2.3.2 The Characteristic Automaton

One part of a bottom-up parser is mysterious. How does the parser know which action it should perform just by looking at the stack? This section demonstrates that it is feasible to do so via the theory of *LR parsing*.

To begin with, an LR parser requires a trivial restriction on its input grammar to simplify its termination condition:

2.12 Definition (Start-separated)
A *start-separated* context-free grammar $G = (N, T, P, S')$ has exactly one production with left-hand side S' of the form $S' \to S$.

$\square$

From here on, all grammars are start-separated.

We start of by formalizing the possible stack contents during a derivation as *viable prefixes* of the grammar.

2.13 Definition
Let $S \overset{*}{\Rightarrow}^{r} \beta A w \Rightarrow^{r} \beta\gamma w$ a rightmost derivation of a context-free grammar G. In this situation, γ is a *handle* of the right-sentential form $\beta\gamma w$ and every prefix of $\beta\gamma$ is a *viable prefix* of G.

As it turns out, the language of viable prefixes of G is a regular language. In the following, we will construct a nondeterministic finite automaton for this language, the *characteristic automaton of G*.

To build the set of states for the characteristic automaton requires to abstract from the actual state of the parser. The proper abstraction is a *context-free item* of the grammar.

2.14 Definition (context-free item)
The set *Items*(G) of context-free items of G consists of all triples of the form $A \to \alpha{\cdot}\beta$ where $A \to \alpha\beta \in P$.

$\square$

Intuitively, an item $A \to \alpha \cdot \beta$ abstracts a parser state $\gamma\alpha \bullet vw$ if there is a rightmost derivation $S \overset{*}{\Rightarrow}^{r} \gamma A w$ and $\beta \overset{*}{\Rightarrow} v$. The formal definition specifies this connections by calling an item *valid*.

2.15 Definition
An item $A \to \alpha \cdot \beta$ is *valid* for viable prefix $\gamma\alpha$ if there is a rightmost derivation $S \overset{*}{\Rightarrow}^{r} \gamma A w \overset{*}{\Rightarrow}^{r} \gamma\alpha\beta w$.

□

Now we can state the automaton that recognizes the set of viable prefixes.

2.16 Definition
Let $G = (N, T, P, S')$ be a context-free grammar. The characteristic NFA of G is $char(G) = (Q, N \cup T, q_0, \delta, F)$ with

- $Q = Items(G)$
- $q_0 = S' \to \cdot S$
- $F = \{A \to \alpha \cdot \mid A \to \alpha \in P\}$
- $\delta(A \to \alpha \cdot X\beta, X) \ni A \to \alpha X \cdot \beta$
- $\delta(A \to \alpha \cdot B\beta, \epsilon) \ni B \to \cdot\gamma$ if $B \to \gamma \in P$.

9 Example
Construct $char(G)$ for the grammar of arithmetic expressions. The table below omits items without transitions.

item \ symbol	2	x	(	)	+	*	T	E	F
$[S \to \cdot T]$							$[S \to T\cdot]$		
$[T \to \cdot E]$								$[T \to E\cdot]$	
$[T \to \cdot T\text{+}E]$							$[T \to T \cdot \text{+}E]$		
$[E \to \cdot F]$									$[E \to F\cdot]$
$[E \to \cdot E{*}F]$								$[E \to E \cdot {*}F]$	
$[F \to \cdot 2]$	$[F \to 2\cdot]$								
$[F \to \cdot \mathrm{x}]$		$[F \to \mathrm{x}\cdot]$							
$[F \to \cdot (T)]$			$[F \to (\cdot T)]$						
$[F \to (\cdot T)]$							$[F \to (T \cdot)]$		
$[T \to T \cdot \text{+}E]$					$[T \to T\text{+} \cdot E]$				
$[E \to E \cdot {*}F]$						$[E \to E{*} \cdot F]$			
$[F \to (T \cdot)]$				$[F \to (T)\cdot]$					
$[T \to T\text{+} \cdot E]$								$[T \to T\text{+}E\cdot]$	
$[E \to E{*} \cdot F]$									$[E \to E{*}F\cdot]$

and the ε transitions:

$$\begin{array}{lcl}
[S \to \cdot T] & \overset{\varepsilon}{\mapsto} & [T \to \cdot E], [T \to \cdot T\text{+}E] \\
[T \to \cdot E] & \overset{\varepsilon}{\mapsto} & [E \to \cdot F], [E \to \cdot E{*}F] \\
[T \to \cdot T\text{+}E] & \overset{\varepsilon}{\mapsto} & [T \to \cdot E], [T \to \cdot T\text{+}E] \\
[E \to \cdot F] & \overset{\varepsilon}{\mapsto} & [F \to \cdot 2], [F \to \cdot \mathrm{x}], [F \to \cdot (T)] \\
[E \to \cdot E{*}F] & \overset{\varepsilon}{\mapsto} & [E \to \cdot F], [E \to \cdot E{*}F] \\
[F \to (\cdot T)] & \overset{\varepsilon}{\mapsto} & [T \to \cdot E], [T \to \cdot T\text{+}E] \\
[T \to T\text{+} \cdot E] & \overset{\varepsilon}{\mapsto} & [E \to \cdot F], [E \to \cdot E{*}F] \\
[E \to E{*} \cdot F] & \overset{\varepsilon}{\mapsto} & [F \to \cdot 2], [F \to \cdot \mathrm{x}], [F \to \cdot (T)]
\end{array}$$

□

2.3.3 LR(0) Parsing

As a first step towards a deterministic parsing engine, we construct a deterministic version of the characteristic automaton.

2.17 Definition (Prediction and Closure)
Each state $q \in \mathcal{P}(Items(G))$ has an associated set of *predict items*:

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

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

$$A \to \alpha \cdot B\beta \Downarrow B \to \cdot\delta$$

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

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

denotes the closure of a state q.

□

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.

2.18 Definition
The set of *LR states* for grammar G is $\textit{LR-state}(G) = \{q \in \mathcal{P}(\textit{Items}(G)) \mid q = \overline{q}\}$.

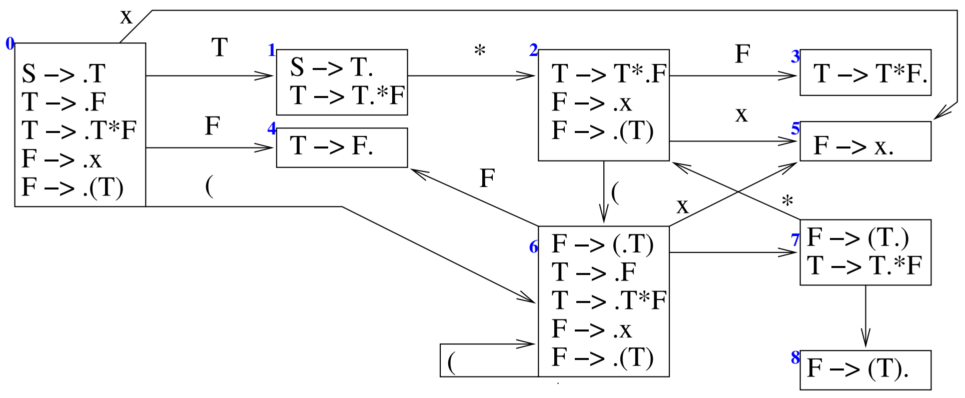
With these definitions, it is straightforward to directly define the deterministic version of the characteristic automaton.

2.19 Definition
The *LR-DFA of G* is $(Q, N \cup T, \text{goto}, q_0, F)$ where

- $Q = \textit{LR-state}(G)$
- $\text{goto}(q, X) = \overline{\{A \to \alpha X \cdot \beta \mid A \to \alpha \cdot X\beta \in q\}}$
- $q_0 = \overline{\{S' \to \cdot S\}}$
- $F = \{q \in Q \mid A \to \alpha\cdot \in q\}$

10 Example
The LR-DFA for the grammar of arithmetic expressions is fairly large, so we content ourselves with a grammar for a sublanguage.

$$S \to T \qquad T \to F \qquad T \to T{*}F \qquad F \to \mathtt{x} \qquad F \to (T)$$



The parser is driven directly from the LR-DFA of the grammar.

Putting the parts together results in the definition of a nondeterministic parser for G. The parser consists of two mutually recursive functions parse and shift. Function parse has three choices. If the current state q (which is always on top of the stack) contains a reduce item $A \to \alpha\cdot$, then it removes $|\alpha|$ symbols from the stack and attempts to shift the left-hand side A from that state. If it makes sense to shift the next symbol, it does so. Finally, if the input is depleted and there is a reduce item for the start production, then it signals success. The shift function just changes state by invoking goto on the top state of the stack and pushing the resulting new state.

$$
\begin{array}{lcl}
\mathrm{parse}(\mathit{stack}, w) & = & \textbf{let } q = \mathrm{top}(\mathit{stack}) \textbf{ in} \\
 & & \quad \bigvee\{\mathrm{shift}(\mathtt{a}, \mathit{stack}, w') \mid w = \mathtt{a}w', A \to \alpha \cdot \mathtt{a}\beta \in q\} \\
 & & \vee \quad \bigvee\{\mathrm{shift}(A, \mathrm{pop}(|\alpha|, \mathit{stack}), w) \mid A \to \alpha\cdot \in q\} \\
 & & \vee \quad w = \epsilon \wedge S' \to S\cdot \in q \wedge |\mathit{stack}| = 1 \\
\\
\mathrm{shift}(X, \mathit{stack}, w) & = & \textbf{let } q = \mathrm{top}(\mathit{stack}) \textbf{ in} \\
 & & \mathrm{parse}(\mathrm{push}(\mathrm{goto}(q, X), \mathit{stack}), w)
\end{array}
$$

Again, the specification is nondeterministic, in general. But we obtain a good idea of the sources of nondeterminism by examining the current state q of the LR-DFA. Specifically, the nondeterminism is caused by two kinds of *unsuitable states* in the LR-DFA:

1. State q has a **reduce-reduce conflict** if it contains two different *reduce items* $A \to \alpha\cdot$ and $B \to \beta\cdot$.

2. State q has a **shift-reduce conflict** if it contains a reduce item $A \to \alpha\cdot$ and a shift item $B \to \beta \cdot \mathtt{a}\gamma$ on a terminal symbol $\mathtt{a}$.

However, the LR-DFA for a grammar G may be free of conflicts already. In this case, G is a *LR(0) grammar* because it is amenable to deterministic LR-parsing without any lookahead.

11 Example
The grammar for arithmetic expressions has no reduce-reduce conflicts, but there are shift-reduce conflicts in the following states:

$$
\begin{array}{ll}
\{T \to E\cdot, & E \to E \cdot {*}F\} \\
\{T \to T{+}E\cdot, & E \to E \cdot {*}F\} \\
\{S' \to T\cdot, & T \to T \cdot {+}E\}
\end{array}
$$

The grammar for simplified arithmetic expressions from Example 10 has a shift-reduce conflict in state 1:

$$\{S \to T\cdot, T \to T \cdot {*}F\}.$$

Hence, neither grammar is an LR(0) grammar.

2.3.4 Implementation of LR(0) Parsing

This subsection contains a complete implementation of a (nondeterministic) stack-based LR(0) parser. This parser computes the predict and closure functions on the fly as follows.

```
(* LR(0) closure of an item set *)
let closure g items =
  let rec worker prestate worklist =
    match worklist with
      [] ->
        prestate
    | item::items ->
        match Item.rhs_rest item with
          Cfg.NT (n)::_ ->
```

```
            let productions_with_n = Grammar.productions_with_lhs g n in
            let candidates = List.map Item.initial productions_with_n in
            let newitems = filter (function cand -> List.mem cand prestate) candidates in
            worker (newitems@prestate) (newitems@worklist)
        | _ ->
            worker prestate items
  in
  worker items items
```

The next function tests if the parser can shift a (terminal or nonterminal) symbol on an item.

```
(* canshift : ('n,'t) symbol -> ('n,'t,'ext) Item.t -> bool *)
let canshift symbol item =
  match Item.rhs_rest item with
    [] -> false
  | x::_ -> x = symbol
```

The heart of the shift function computes the shifted versions of the items which can be shifted on symbol. The result still needs to be closed with respect to the predict items.

```
(* goto : ('n,'t,'ext) Item.t list -> ('n,'t,'ext) Item.t list *)
let goto state symbol =
  List.map Item.shift (filter (canshift symbol) state)
```

The main parser function, accept, contains the above specified functions, parse and shift. It first prepares the initial state by closing over the predict item of the start production and then leaves the work to parse, which works exactly as specified.

```
(* accept : ('n,'t,'ext) Cfg.grammar -> 't list -> bool *)
let accept g inp =
  let start_production::_ = Grammar.productions_with_lhs g (Cfg.start g) in
  let initial_state = closure g [Item.initial start_production] in
  let is_final_state state =
    match state with
      [item] ->
        Item.production item = start_production && Item.complete item
    | _ ->
        false
  in
  (* parse : ('n,'t,'ext) Item.t list list -> 't list -> bool *)
  let rec parse stack inp =
    let state :: stack_rest = stack in
    (match inp with
      t :: inp_rest ->
        List.exists (canshift (Cfg.T (t))) state
          && shift (Cfg.T (t)) stack inp_rest
    | [] ->
        is_final_state state && stack_rest = [])
    ||
      List.exists
        (function reducible_item ->
          shift
            (Cfg.NT (Cfg.lhs (Item.production reducible_item)))
          (drop (Item.position reducible_item) stack)
          inp)
      (filter Item.complete state)
(* shift : ('n,'t) symbol -> ('n,'t,'ext) Item.t list list -> 't list -> bool *)
and shift symbol stack inp =
  let state::_ = stack in
  parse (closure g (goto state symbol)::stack) inp

in
parse [initial_state] inp
```

The auxiliary function `List.exists` takes a predicate and a list. It returns `true` if there is a list element which makes the predicate true. The uses of `List.exists` correspond to the large disjunctions in the specification.

2.3.5 LR(k) Parsing

The standard medicine for resolving conflicts (and hence nondeterminism) is to add lookahead to the parsing engine. This section first looks at the canonical way of adding lookahead, which turns out to be very expensive. Subsequent subsections consider simpler and more efficient means of adding lookahead information.

2.20 Definition
Let $G = (N, T, P, S')$ be a start separated context-free grammar. G is an *LR(k) grammar* if

- $S' \overset{*}{\Rightarrow}^{r} \alpha A w \Rightarrow^{r} \alpha\beta w$ and
- $S' \overset{*}{\Rightarrow}^{r} \gamma B u \Rightarrow^{r} \alpha\beta v$ and
- $w_{|k} = v_{|k}$

implies that $\alpha = \gamma$, $A = B$, and $u = v$.

LR(k) parsing is a very strong formalism as the following facts demonstrate.

1. If G is an LL(k) grammar, then G is an LR(k) grammar.
2. If L is a deterministic context-free language, then L has a LR(1) grammar. In particular: If L has an LR(k) grammar, then it also has an LR(1) grammar.

The definition of the LR(k)-DFA encompasses essentially the same steps as the LR-DFA. The main difference is the extension of items by lookahead sets.

2.21 Definition (LR(k) item, LR(k) state)
Let G be a start separated, context-free grammar. The set *LR(k)-Items*(G) contains all quadruples of the form $A \to \alpha \cdot \beta\ (L)$ where $A \to \alpha\beta$ is a production of G and $L \subseteq T^{\leq k}$. The set L indicates the set of lookahead strings for which the item is valid.

If the lookahead is not used (or $k = 0$), it is omitted. A *predict item* has the form $A \to \cdot\alpha\ (L)$.

□

The definition of a valid item extends smoothly. The important point in the definition is that the lookahead does not refer to the position of the dot in the item but rather describes the symbols that may follow the item's nonterminal in a derivation for which the item is valid.

2.22 Definition
An item $A \to \alpha \cdot \beta\ (L)$ is *valid* for viable prefix $\gamma\alpha$ if there is a rightmost derivation $S \overset{*}{\Rightarrow}^{r} \gamma A w \overset{*}{\Rightarrow}^{r} \gamma\alpha\beta w$ and $w_{|k} \in L$.

□

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

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

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

$$A \to \alpha \cdot B\beta\ (L) \Downarrow B \to \cdot\delta\ (\text{first}_k(\beta L)).$$

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

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

denotes the closure of a state q.

□

The state set of the LR(k)-DFA is again formed from the set of LR(k) states.

2.24 Definition
The set of *LR(k) states* for grammar G is

$$\textit{LR(k)-state}(G) = \{q \subseteq \textit{LR(k)-Items}(G) \mid q = \overline{q}\}$$

The LR(k)-DFA just adds the treatment of lookahead to the LR-DFA: the goto function preserves the lookaheads, the lookahead of the start production is $\{\epsilon\}$, and the final states are not affected by lookahead at all.

2.25 Definition
The *LR(k)-DFA of G* is $(Q, N \cup T, \text{goto}, q_0, F)$ where

- $Q = \textit{LR(k)-state}(G)$
- $\text{goto}(q, X) = \overline{\{A \to \alpha X \cdot \beta\ (L) \mid A \to \alpha \cdot X\beta\ (L) \in q\}}$
- $q_0 = \overline{\{S' \to \cdot S\ (\{\epsilon\})\}}$
- $F = \{q \in Q \mid A \to \alpha \cdot\ (L) \in q\}$

The parsing engine itself requires very little modification.

```
parse($stack, w$)
    =  let $q$ = top($stack$) in
           $\bigvee\{\text{shift}(A, \text{pop}(|\alpha|, stack), w) \mid A \to \alpha \cdot\ (L) \in q, w_{|k} \in L\}$
       ∨   $\bigvee\{\text{shift}(\mathtt{a}, stack, w') \mid w = \mathtt{a}w', A \to \alpha \cdot \mathtt{a}\beta\ (L) \in q, w_{|k} \in \text{first}_k(\mathtt{a}\beta L)\}$
       ∨   $w = \epsilon \wedge S' \to S \cdot\ (\{\epsilon\}) \in q$

shift($X, stack, w$)
    =  let $q$ = top($stack$) in
           parse(push(goto($q, X$), $stack$), $w$)
```

The following conflicts are possible (and lead to nondeterminism) in such a parser:

1. State q has a **reduce-reduce conflict** if it contains two different *reduce items* $A \to \alpha \cdot\ (L)$ and $B \to \beta \cdot\ (M)$ such that $L \cap M \neq \emptyset$.
2. State q has a **shift-reduce conflict** if it contains a reduce item $A \to \alpha \cdot\ (L)$ and a shift item $B \to \beta \cdot \mathtt{a}\gamma\ (M)$ on a terminal symbol $\mathtt{a}$ such that $L \cap \mathrm{first}_k(\mathtt{a}\gamma\ M) \neq \emptyset$.

The LR(k)-DFA does not contain reachable states that exhibit one of the conflicts if and only if the grammar is LR(k).

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

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

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

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

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

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

The parsing engine for an SLR(k) parser is identical to the one for the LR(k) parser. The only difference is in the computation of the lookahead sets. The effects of using SLR(k) instead of LR(k) are as expected: generation time and size decrease, often dramatically for realistic grammars.

12 Example
The grammar for arithmetic expressions is an SLR(1) grammar. To see this, we review the unsuitable states of its LR-DFA and find that all conflicts can be resolved by adding SLR(1) lookahead sets:

$$\begin{array}{ll} \{T \to E\cdot, & E \to E \cdot {*}F\} \\ \{T \to T{+}E\cdot, & E \to E \cdot {*}F\} \\ \{S' \to T\cdot, & T \to T \cdot {+}E\} \end{array}$$

According to SLR(1), the lookahead sets for the reduce items in question are the follow_1 sets of T and S'. Examination of the grammar yields that

$$\begin{array}{lcl} \mathrm{follow}_1(T) & = & \{\epsilon, \mathtt{)}, \mathtt{+}\} \\ \mathrm{follow}_1(S') & = & \{\epsilon\} \end{array}$$

In the first and second state, the conflict is resolved because the set $\mathrm{follow}_1(T)$ does not contain the single symbol $\mathtt{*}$ on which the state can shift. In the third state, the conflict is also resolved because $\mathrm{follow}_1(S')$ does not contain $\mathtt{+}$.

□

2.3.7 LALR Lookahead Computation

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].

Here is the definition for the LALR(1) lookahead of an LR(0)-item. The main novelty here is that the lookahead depends on the state of the automaton, too.

2.26 Definition
Let $(Q, N \cup T, \text{goto}, q_0, F)$ be the LR-DFA for a start-separated grammar G. Let further $q \in Q$ and $\mathcal{I} = A \to \alpha \cdot \beta \in q$. The *LALR(1) lookahead of* $\mathcal{I}$ *in* q is

$$LA_1(q, A \to \alpha \cdot \beta) = \{w_{|1} \mid S \overset{*}{\Rightarrow}^{r} \gamma A w, q = \text{goto}^*(q_0, \gamma\alpha)\}$$

The main interest is, of course, in the lookahead for the reduce items, that is, in $LA_1(q, A \to \alpha\cdot)$, but the general definition makes it easier to find an obviously computable definition for the lookahead sets.

This definition involves a quantification over all derivations, which makes it pretty hard to implement. Following the calculation in Wilhelm's textbook [WM92], it can be simplified as follows.

The first observation is that the lookahead for an item with the dot in the middle can be expressed in terms of lookaheads for predict items, that is, items with the dot at the left end of their right hand side. This transformation exploits the factoring of goto^* with respect to its input word, that is, $\text{goto}^*(q_0, \gamma\alpha) = \text{goto}^*(\text{goto}^*(q_0, \gamma), \alpha)$.

$$\begin{array}{rl} & LA_1(q, A \to \alpha \cdot \beta) \\ = & \{w_{|1} \mid S \overset{*}{\Rightarrow}^{r} \gamma A w, q = \text{goto}^*(q_0, \gamma\alpha)\} \\ = & \{w_{|1} \mid S \overset{*}{\Rightarrow}^{r} \gamma A w, q' = \text{goto}^*(q_0, \gamma), q = \text{goto}^*(q', \alpha)\} \\ = & \bigcup_{q = \text{goto}^*(q', \alpha)} LA_1(q', A \to \cdot\alpha\beta) \end{array}$$

The goal is now to express the lookahead sets of predict items in terms of lookahead sets of other predict items, thus giving raise to a system of equations on lookahead sets. We write $\widehat{LA}_1(q, A) = LA_1(q, A \to \cdot\alpha)$, noticing that the lookahead set is independent of α.

A predict item can either be the start item $S' \to \cdot S$, in which case the lookahead set is $\{\epsilon\}$ because the input word should be exhausted after something has been derived from S, or the item $A \to \cdot\alpha$ is in q due to the closure operation. In the second case, it must have been added by the predict operation so that the state q must contain one or more items of the form $B \to \beta \cdot A\gamma$. These two cases give rise to the following equations:

$$\begin{array}{rcl}
\widehat{LA}_1(q_0, S') & = & \{w_{|1} \mid S' \overset{*}{\Rightarrow}^r \gamma S' w, q = \text{goto}^*(q_0, \gamma\alpha)\} \\
& & \text{because } \gamma = \epsilon,\ w = \epsilon, \text{ and } q = q_0 \\
& = & \{\epsilon\} \\
\\
\widehat{LA}_1(q, A) & = & \{(uv)_{|1} \mid S' \overset{*}{\Rightarrow}^r \gamma\alpha A uv, q = \text{goto}^*(q_0, \gamma\alpha)\} \\
& = & \{(uv)_{|1} \mid \ S' \overset{*}{\Rightarrow}^r \gamma B v \Rightarrow^r \gamma\alpha A\beta v \overset{*}{\Rightarrow}^r \gamma\alpha A uv, \\
& & \qquad q = \text{goto}^*(q_0, \gamma\alpha), B \to \alpha \cdot A\beta \in q\} \\
& = & \{(uv)_{|1} \mid \ S' \overset{*}{\Rightarrow}^r \gamma B v \Rightarrow^r \gamma\alpha A\beta v, \\
& & \qquad u \in \text{first}_1(\beta), q = \text{goto}^*(q_0, \gamma\alpha), B \to \alpha \cdot A\beta \in q\} \\
& = & \{(uv)_{|1} \mid \ S' \overset{*}{\Rightarrow}^r \gamma B v, \\
& & \qquad u \in \text{first}_1(\beta), v \in \widehat{LA}_1(q', B), \\
& & \qquad q = \text{goto}^*(q', \alpha), q' = \text{goto}^*(q_0, \gamma), B \to \alpha \cdot A\beta \in q\} \\
& = & \bigcup_{B \to \alpha \cdot A\beta \in q} \bigcup_{q = \text{goto}^*(q', \alpha)} (\text{first}_1(\beta)\widehat{LA}_1(q', B))_{|1}
\end{array}$$

This system of equations has (at most) $|Q| \times |N|$ variables and it can be solved by fixpoint iteration. More clever, essentially linear-time algorithms exist and are documented in the literature.

A solution of the system of equations for $\widehat{LA}_1$ yields the desired result, the lookahead sets for the reduce items, as follows:

$$LA_1(q, A \to \alpha\cdot) = \bigcup_{\text{goto}^*(q', \alpha) = q} \widehat{LA}_1(q', A)$$

2.4 Output of a Parser

A parser which just outputs yes or no is not of much use in a compiler. Hence, we augment the parsing formalism with a notion of syntax representation which can serve as parser output.

2.27 Definition (Syntax representation)
Let T be the terminal alphabet. A parser *parse* : $T^* \to D$ generates a *syntax representation* from a set D if there is a function *unparse* : $D \to T^*$ such that *parse* $\circ$ *unparse* $= id_D$.

□

The main intention of the definition is to provide the minimum requirements for D. If the parser is based on context-free grammars, then the natural choice for D would be the set of derivation trees of the grammar.

In fact, both styles of parsers can construct a derivation tree during parsing. The idea is consider a production $A_0 \to w_0 A_1 w_1 \dots A_n w_n$ as a tree constructor function that takes n derivation trees for nonterminals $A_1, \dots, A_n$ and returns a derivation tree for nonterminal A_0.

A recursive-descent parser needs no additional structure to do so. The function $[A_0]$, for parsing strings derived from A_0, just applies the appropriate constructor to the derivation trees obtained from the calls to $[A_1], \dots, [A_n]$ and returns he resulting tree along with the rest of the input.

A shift-reduce parser would be able to build the derivation tree on its stack. For clarity, however, we extend the generic shift-reduce parser with an output stack. The idea is then to apply the tree constructor function for $A_0 \to w_0 A_1 w_1 \dots A_n w_n$ to the topmost n entries of the output stack and replace them with the resulting tree.

13 Example
Let's revisit Example 8 with output generation. A configuration is now a triple $\gamma \bullet w \rhd \Pi$ where Π is a stack of derivation trees with entries separated by ::. We write $[A \to \alpha]$ for the tree constructor of $A \to \alpha$. We put the argument trees in parentheses, but omit them if there are none.

$+\bullet$	$\texttt{2+x*x}$	$\rhd$		shift
$2+\bullet$	$\texttt{+x*x}$	$\rhd$		reduce $F \to 2$
$F+\bullet$	$\texttt{+x*x}$	$\rhd$	$[F \to 2]$	reduce $E \to F$
$E+\bullet$	$\texttt{+x*x}$	$\rhd$	$[T \to E]([F \to 2])$	reduce $T \to E$
$T+\bullet$	$\texttt{+x*x}$	$\rhd$	$[T \to E]([F \to 2])$	shift
$T\texttt{+}+\bullet$	$\texttt{x*x}$	$\rhd$	$[T \to E]([F \to 2])$	shift
$T\texttt{+x}+\bullet$	$\texttt{*x}$	$\rhd$	$[T \to E]([F \to 2])$	reduce $F \to \texttt{x}$
$T\texttt{+}F+\bullet$	$\texttt{*x}$	$\rhd$	$[T \to E]([F \to 2]) :: [F \to \texttt{x}]$	reduce $F \to E$
$T\texttt{+}E+\bullet$	$\texttt{*x}$	$\rhd$	$[T \to E]([F \to 2]) :: [E \to F]([F \to \texttt{x}])$	shift
$T\texttt{+}E\texttt{*}+\bullet$	$\texttt{x}$	$\rhd$	$[T \to E]([F \to 2]) :: [E \to F]([F \to \texttt{x}])$	shift
$T\texttt{+}E\texttt{*x}+\bullet$		$\rhd$	$[T \to E]([F \to 2]) :: [E \to F]([F \to \texttt{x}])$	reduce $F \to \texttt{x}$
$T\texttt{+}E\texttt{*}F+\bullet$		$\rhd$	$[T \to E]([F \to 2]) :: [E \to F]([F \to \texttt{x}]) :: [F \to \texttt{x}]$	reduce $E \to E\texttt{*}F$
$T\texttt{+}E+\bullet$		$\rhd$	$[T \to E]([F \to 2]) :: [E \to E\texttt{*}F]([E \to F]([F \to \texttt{x}]), [F \to \texttt{x}])$	reduce $T \to T\texttt{+}E$
$T+\bullet$		$\rhd$	$[T \to T\texttt{+}E]([T \to E]([F \to 2]), [E \to E\texttt{*}F]([E \to F]([F \to \texttt{x}]), [F \to \texttt{x}]))$	success

The example shows clearly that derivation trees may contain information which is not relevant for the meaning of the phrase. In this case, the chain productions $E \to F$ and $T \to E$ carry no meaning but they are cluttering the derivation tree. In fact, the nonterminals E and T are only present to model operator precedences. In other grammars, there may be nonterminals and productions to make the grammar palatable to the chosen parsing technology, as in the expression grammar transformed for use with an LL(1) parser.

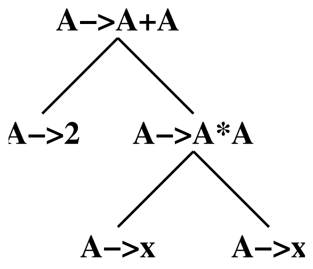
However, the definition of a syntax representation leaves the freedom to choose a more abstract representation that elides extra nonterminals and productions. Such a representation, which only carries semantically relevant information, is called an *abstract syntax* representation or *abstract syntax tree* (AST).

For convenience, abstract syntax is often defined by a grammar. This grammar is usually unsuitable for parsing (in fact, it is often ambiguous), but that is quite ok because the interest is only in the derivation trees of the grammar. In a language like OCaml, abstract syntax fits exactly with algebraic datatypes.

14 Example
Here is a grammar suitable for defining the abstract syntax of arithmetic expressions:

$$A \to 2 \mid \texttt{x} \mid A\texttt{+}A \mid A\texttt{*}A$$

The intended derivation tree for `2+x*x` is



An OCaml type definition expresses the same structure more concisely and provides a notation for the trees at the same time.

```
type Expr = Two | Ex | Add of Expr * Expr | Mul of Expr * Expr

Add (Two, Mul (Ex, Ex))
```

This expression provides a description of the input string `2+x*x` which captures all ingredients for defining its meaning precisely. Moreover, OCaml provides notation and techniques for defining functions to further process these trees.

2.5 Recursive-Ascent Parsing

2.5.1 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.

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.

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 $q_0 = \{S' \to \cdot S\}$.

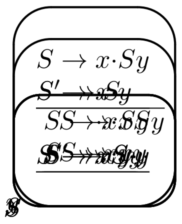


Figure 2.2: LR state diagram for $S' \to S, S \to xSy, S \to xy$

All parser states are results of applications of goto. Figure 2.2 shows an example state transition diagram omitting lookahead.

2.5.2 Continuation-Based Recursive Ascent Parsing

It is possible to express LR parsing in a continuation-based form [Spe94] that elides the explicit stack. Instead of keeping the entire stack, the parser just keeps its topmost entries around. To do so, it has to know just how many entries may have to be popped from the stack at one time.

2.28 Definition (Next terminals)

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

□

2.29 Definition (Active symbols)
Each state q has an associated *number of active symbols*, nactive(q):

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

□

When the parser is in state q, then 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].

2.30 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.

2.31 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:

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

$$
\begin{array}{lll}
\multicolumn{3}{l}{[q](w, c_1, \ldots, c_{\text{nactive}(q)}) :=} \\
\quad \textbf{letrec} & c_0(X, w) = & \\
& \quad [\text{goto}(q, X)]\,(w, c_0, c_1, \ldots, c_{\text{nactive}(\text{goto}(q,X))-1}) & \\
\quad \textbf{in} & A \to \alpha \cdot\ (u) \in \overline{q} \wedge w_{|k} = u & \rhd c_{|\alpha|}(A, w) \\
\quad | & w = \mathtt{a}w' \wedge A \to \alpha \cdot \mathtt{a}\beta \in q & \rhd c_0(x, w')
\end{array}
$$

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

2.5.3 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 $\mathtt{n} \to \nu(v)$ where $v \in \mathrm{first}_k(\mathtt{la\_suffix})$:

```
  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 2.25:

```
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 2.29:

```
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 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
```

2.6 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:

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

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.

Bibliography

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[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.

[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.

[FWH01] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* MIT Press and McGraw-Hill, 2nd edition, 2001.

[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.

[WM96] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau — Theorie, Konstruktion, Generierung — 2 Auflage.* Lehrbuch. Springer-Verlag, Berlin, Heidelberg, 1996.