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.

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

arithmetic expressions from Example 4.

$\bullet 2 + \mathtt{x} * \mathtt{x}$	shift
$2 \bullet + \mathtt{x} * \mathtt{x}$	reduce $F \to 2$
$F \bullet + \mathtt{x} * \mathtt{x}$	reduce $E \to F$
$E \bullet + \mathtt{x} * \mathtt{x}$	reduce $T \to E$
$T \bullet + \mathtt{x} * \mathtt{x}$	shift
$T + \bullet\, \mathtt{x} * \mathtt{x}$	shift
$T + \mathtt{x} \bullet * \mathtt{x}$	reduce $F \to \mathtt{x}$
$T + F \bullet * \mathtt{x}$	reduce $F \to E$
$T + E \bullet * \mathtt{x}$	shift
$T + E * \bullet\, \mathtt{x}$	shift
$T + E * \mathtt{x} \bullet$	reduce $F \to \mathtt{x}$
$T + E * F \bullet$	reduce $E \to E * F$
$T + E \bullet$	reduce $T \to T + E$
$T \bullet$	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:

2.13 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 $S' \to S$.*

□

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.

2.14 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.

2.15 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 $A \to \alpha\beta \in P$.

□

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*.

2.16 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.

2.17 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* $B \to \gamma \in P$.

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

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

and the ε transitions:

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

□

2.3.3 LR(0) Parsing

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

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

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

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 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 predict(q) are exactly those at the end of leftmost-symbol derivations starting from items in q.

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

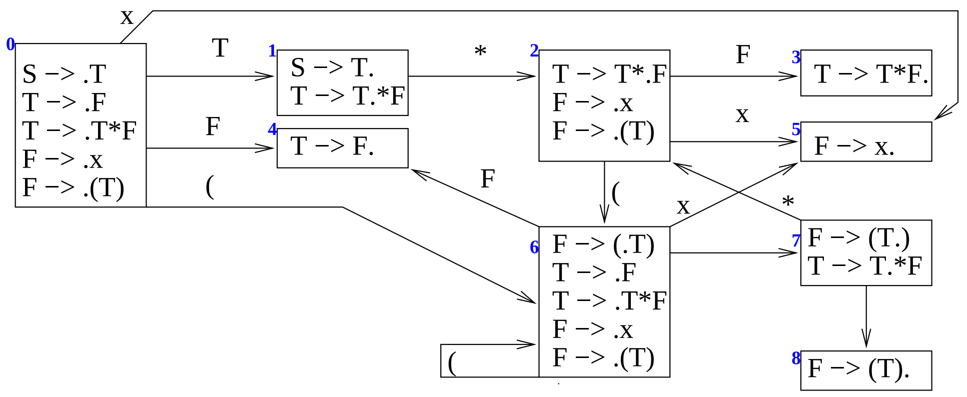
With these definitions, it is straightforward to directly define the deterministic version of the characteristic automaton.

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

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

9 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.

10 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 9 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.

2.21 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.

2.22 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 $L \subseteq T^{\le k}$. 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.

2.23 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 $w_{|k} \in L$.*

□

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

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

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.

2.25 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.

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

shift($X$, $stack$, $w$)
    $=$ let $q = \text{top}(stack)$ 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 $q_0^{(0)}, \ldots, q_n^{(0)}$ has states closures $q_0, \ldots, q_n$. In contrast to the LR(k) parser, the SLR(k) automaton has the following states:

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

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

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

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.

11 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

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

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

□

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.

2.27 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:

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

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.

2.28 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 $A_0 \to w_0A_1w_1 \dots A_nw_n$ 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 $A_0 \to w_0A_1w_1 \dots A_nw_n$ to the topmost n entries of the output stack and replace them with the resulting tree.

12 Example
Let's revisit Example 7 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.

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

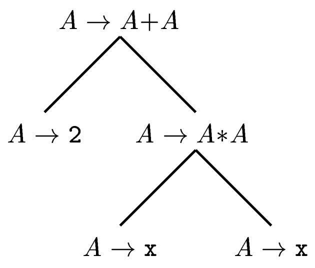
However, the definition of a syntax representation leaves the freedom to choose a more abstract representation that elides extra nonterminals and productions. Such a representation, which only carries semantically relevant information, is called an *abstract syntax* representation or *abstract syntax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

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

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

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.