Lenguajes Formales y Compiladores

Syntax Analysis

Sergio Ramírez Rico

UNIVERSIDAD EAFIT

Área Ciencias Fundamentales
Escuela de Ciencias Aplicadas e Ingeniería

Motivation

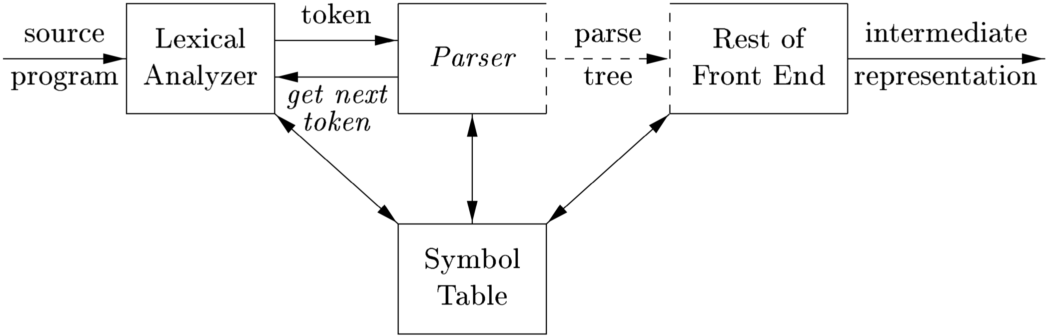


Figure 4.1 Aho et al. 2006.

Table of Contents

Recursive-Descent Parsing

Takeaway

This technique uses backtracking to find the correct production to be applied to derive the input string.

Recursive-Descent Parsing

Algorithm Recursive Procedure for Parsing

Read input left to right. Let a be the current input symbol
procedure $\textsc{Recursive-parser}(A)$ ▷ Starts with S
    Choose an A-production: $A \to x_1 x_2 \cdots x_k$ ▷ $x_i \in N \cup \Sigma$
    for $i := 1$ to k **do**
        if $x_i \in N$ **then**
            $\textsc{Recursive-parser}(x_i)$
        else if $x_i = a$ **then**
            advance to the next input symbol
        else
            `ERROR`

Example

Recursive-Descent Parsing

Consider the grammar

$$S \to cAd$$
$$A \to ab \mid a$$

Construct a parse tree top-down for the input string $w = cad$.

Table of Contents

Takeaway

Takeaway

First and Follow are auxiliary sets that allow us to find the terminal symbols that start a string and the terminal symbols that follow a given nonterminal symbol, respectively.

Note: We assume that every string ends with the symbol \$.

First

Definition (First)

Let $G = (N, \Sigma, P, S)$ be a grammar.
Define $\textsc{First}(\alpha)$, where $\alpha \in (N \cup \Sigma)^*$, to be the set of terminals that begin strings derived from α.

First

Definition (First)

Let $G = (N, \Sigma, P, S)$ be a grammar.
Define $\textsc{First}(\alpha)$, where $\alpha \in (N \cup \Sigma)^*$, to be the set of terminals that begin strings derived from α.

Remark: We allow ε to be in First sets under certain conditions.

First of a Symbol (I)

Compute $\textsc{First}(x)$

To compute $\textsc{First}(x)$ for all $x \in N \cup \Sigma$, apply the following rules until no more terminals or ε can be added to any First set.

First of a Symbol (I)

Compute $\textsc{First}(x)$

To compute $\textsc{First}(x)$ for all $x \in N \cup \Sigma$, apply the following rules until no more terminals or ε can be added to any First set.

1. If $x \in \Sigma$, then $\textsc{First}(x) := \{x\}$.

First of a Symbol (I)

Compute $\textsc{First}(x)$

To compute $\textsc{First}(x)$ for all $x \in N \cup \Sigma$, apply the following rules until no more terminals or ε can be added to any First set.

1. If $x \in \Sigma$, then $\textsc{First}(x) := \{x\}$.
2. If $x \in N$ and $x \rightarrow y_1 y_2 \cdots y_i \cdots y_k$ is a production for some $k \geq 1$, then:

First of a Symbol (I)

Compute $\textsc{First}(x)$

To compute $\textsc{First}(x)$ for all $x \in N \cup \Sigma$, apply the following rules until no more terminals or ε can be added to any First set.

1. If $x \in \Sigma$, then $\textsc{First}(x) := \{x\}$.

2. If $x \in N$ and $x \to y_1 y_2 \cdots y_i \cdots y_k$ is a production for some $k \geq 1$, then:

    2.1 For each $1 \leq i \leq k$, if $\varepsilon \in \textsc{First}(y_j)$ for all $1 \leq j \leq i-1$, then $\textsc{First}(x) := \textsc{First}(x) \cup \textsc{First}(y_i) - \{\varepsilon\}$.

First of a Symbol (I)

Compute $\textsc{First}(x)$

To compute $\textsc{First}(x)$ for all $x \in N \cup \Sigma$, apply the following rules until no more terminals or ε can be added to any First set.

1. If $x \in \Sigma$, then $\textsc{First}(x) := \{x\}$.
2. If $x \in N$ and $x \to y_1 y_2 \cdots y_i \cdots y_k$ is a production for some $k \geq 1$, then:
   2.1 For each $1 \leq i \leq k$, if $\varepsilon \in \textsc{First}(y_j)$ for all $1 \leq j \leq i-1$, then $\textsc{First}(x) := \textsc{First}(x) \cup \textsc{First}(y_i) - \{\varepsilon\}$.

   2.2 If $\varepsilon \in \textsc{First}(y_j)$ for all $1 \leq j \leq k$, then $\textsc{First}(x) := \textsc{First}(x) \cup \{\varepsilon\}$.

First of a Symbol (I)

Compute $\textsc{First}(x)$

To compute $\textsc{First}(x)$ for all $x \in N \cup \Sigma$, apply the following rules until no more terminals or ε can be added to any First set.

1. If $x \in \Sigma$, then $\textsc{First}(x) := \{x\}$.

2. If $x \in N$ and $x \to y_1 y_2 \cdots y_i \cdots y_k$ is a production for some $k \geq 1$, then:
   2.1 For each $1 \leq i \leq k$, if $\varepsilon \in \textsc{First}(y_j)$ for all $1 \leq j \leq i-1$, then $\textsc{First}(x) := \textsc{First}(x) \cup \textsc{First}(y_i) - \{\varepsilon\}$.

   2.2 If $\varepsilon \in \textsc{First}(y_j)$ for all $1 \leq j \leq k$, then $\textsc{First}(x) := \textsc{First}(x) \cup \{\varepsilon\}$.

3. If $x \to \varepsilon$ is a production, then $\textsc{First}(x) := \textsc{First}(x) \cup \{\varepsilon\}$.

First of a Symbol (II)

Compute $\textsc{First}(x)$

To compute $\textsc{First}(x)$ for all $x \in N \cup \Sigma$, apply the following rules until no more terminals or ε can be added to any First set.

First of a Symbol (II)

Compute $\textsc{First}(x)$

To compute $\textsc{First}(x)$ for all $x \in N \cup \Sigma$, apply the following rules until no more terminals or ε can be added to any First set.

1. If $x \in \Sigma$, then $\textsc{First}(x) := \{x\}$.

First of a Symbol (II)

Compute $\textsc{First}(x)$

To compute $\textsc{First}(x)$ for all $x \in N \cup \Sigma$, apply the following rules until no more terminals or ε can be added to any First set.

1. If $x \in \Sigma$, then $\textsc{First}(x) := \{x\}$.

2. If $x \in N$ and $x \to y_1 y_2 \cdots y_i \cdots y_k$ is a production for some $k \geq 1$, then:

First of a Symbol (II)

Compute $\textsc{First}(x)$

To compute $\textsc{First}(x)$ for all $x \in N \cup \Sigma$, apply the following rules until no more terminals or ε can be added to any First set.

1. If $x \in \Sigma$, then $\textsc{First}(x) := \{x\}$.
2. If $x \in N$ and $x \to y_1 y_2 \cdots y_i \cdots y_k$ is a production for some $k \geq 1$, then:
   2.1 For each $1 \leq i \leq k$, if $\varepsilon \in \textsc{First}(y_j)$ for all $1 \leq j \leq i-1$, then $\textsc{First}(y_i) - \{\varepsilon\} \subseteq \textsc{First}(x)$.

First of a Symbol (II)

Compute $\textsc{First}(x)$

To compute $\textsc{First}(x)$ for all $x \in N \cup \Sigma$, apply the following rules until no more terminals or ε can be added to any First set.

1. If $x \in \Sigma$, then $\textsc{First}(x) := \{x\}$.
2. If $x \in N$ and $x \to y_1 y_2 \cdots y_i \cdots y_k$ is a production for some $k \geq 1$, then:
   2.1 For each $1 \leq i \leq k$, if $\varepsilon \in \textsc{First}(y_j)$ for all $1 \leq j \leq i-1$, then $\textsc{First}(y_i) - \{\varepsilon\} \subseteq \textsc{First}(x)$.
   2.2 If $\varepsilon \in \textsc{First}(y_j)$ for all $1 \leq j \leq k$, then $\varepsilon \in \textsc{First}(x)$.

First of a Symbol (II)

Compute $\textsc{First}(x)$

To compute $\textsc{First}(x)$ for all $x \in N \cup \Sigma$, apply the following rules until no more terminals or ε can be added to any First set.

1. If $x \in \Sigma$, then $\textsc{First}(x) := \{x\}$.
2. If $x \in N$ and $x \to y_1 y_2 \cdots y_i \cdots y_k$ is a production for some $k \geq 1$, then:
   - 2.1 For each $1 \leq i \leq k$, if $\varepsilon \in \textsc{First}(y_j)$ for all $1 \leq j \leq i-1$, then $\textsc{First}(y_i) - \{\varepsilon\} \subseteq \textsc{First}(x)$.
   - 2.2 If $\varepsilon \in \textsc{First}(y_j)$ for all $1 \leq j \leq k$, then $\varepsilon \in \textsc{First}(x)$.
3. If $x \to \varepsilon$ is a production, then $\varepsilon \in \textsc{First}(x)$.

First of a Symbol

$$
\textsc{First}(x) := \begin{cases} \{x\} & \text{if } x \in \Sigma \\ \textsc{First}(x) \cup \{\varepsilon\} & \text{if } (x \to y_1 y_2 \cdots y_k \wedge \varepsilon \in \bigcap_{j=1}^{k} \textsc{First}(y_j)) \vee x \to \varepsilon \\ \textsc{First}(x) \cup \textsc{First}(y_i) - \{\varepsilon\} & \text{if } x \to y_1 y_2 \cdots y_k \wedge \varepsilon \in \bigcap_{j=1}^{i-1} \textsc{First}(y_j) \end{cases}
$$

First of a String

Compute $\textsc{First}(x_1 x_2 \cdots x_n)$

To compute $\textsc{First}(x_1 x_2 \cdots x_n)$:

First of a String

Compute $\textsc{First}(x_1 x_2 \cdots x_n)$

To compute $\textsc{First}(x_1 x_2 \cdots x_n)$:

1. Add to $\textsc{First}(x_1 x_2 \cdots x_n)$ all non-ε symbols of $\textsc{First}(x_1)$,

Continue on next slide

First of a String

Compute $\textsc{First}(x_1 x_2 \cdots x_n)$

To compute $\textsc{First}(x_1 x_2 \cdots x_n)$:

1. Add to $\textsc{First}(x_1 x_2 \cdots x_n)$ all non-ε symbols of $\textsc{First}(x_1)$,
2. also add the non-ε symbols of $\textsc{First}(x_2)$, if $\varepsilon \in \textsc{First}(x_1)$,

First of a String

Compute $\textsc{First}(x_1 x_2 \cdots x_n)$

To compute $\textsc{First}(x_1 x_2 \cdots x_n)$:

1. Add to $\textsc{First}(x_1 x_2 \cdots x_n)$ all non-ε symbols of $\textsc{First}(x_1)$,
2. also add the non-ε symbols of $\textsc{First}(x_2)$, if $\varepsilon \in \textsc{First}(x_1)$,
3. the non-ε symbols of $\textsc{First}(x_3)$, if $\varepsilon \in \textsc{First}(x_1) \cap \textsc{First}(x_2)$,

First of a String

Compute $\textsc{First}(x_1 x_2 \cdots x_n)$

To compute $\textsc{First}(x_1 x_2 \cdots x_n)$:

1. Add to $\textsc{First}(x_1 x_2 \cdots x_n)$ all non-ε symbols of $\textsc{First}(x_1)$,
2. also add the non-ε symbols of $\textsc{First}(x_2)$, if $\varepsilon \in \textsc{First}(x_1)$,
3. the non-ε symbols of $\textsc{First}(x_3)$, if $\varepsilon \in \textsc{First}(x_1) \cap \textsc{First}(x_2)$,
4. and so on.

First of a String

Compute $\textsc{First}(x_1 x_2 \cdots x_n)$

To compute $\textsc{First}(x_1 x_2 \cdots x_n)$:

1. Add to $\textsc{First}(x_1 x_2 \cdots x_n)$ all non-ε symbols of $\textsc{First}(x_1)$,
2. also add the non-ε symbols of $\textsc{First}(x_2)$, if $\varepsilon \in \textsc{First}(x_1)$,
3. the non-ε symbols of $\textsc{First}(x_3)$, if $\varepsilon \in \textsc{First}(x_1) \cap \textsc{First}(x_2)$,
4. and so on.
5. Add ε to $\textsc{First}(x_1 x_2 \cdots x_n)$ if, for all $1 \leq i \leq n$, $\varepsilon \in \textsc{First}(x_i)$.

First of a String

This can be formalized as:

$$\textsc{First}(x_1 x_2 \cdots x_n) \coloneqq \bigcup_{i=1}^{n} \left\{ \textsc{First}(x_i) - \{\varepsilon\} \;\middle|\; \varepsilon \in \bigcap_{j=1}^{i-1} \textsc{First}(x_j) \right\}$$

Follow

Definition (Follow)

Let $G = (N, \Sigma, S, P)$ be a grammar.
Define $\textsc{Follow}(A)$, where $A \in N$, to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is:

$$\{a \in \Sigma \mid S \xrightarrow{*} \alpha A a \beta \text{ for some } \alpha, \beta \in (N \cup \Sigma)^*\}$$

Computing Follow

Compute $\textsc{Follow}(A)$

Let $G = (N, \Sigma, S, P)$ be a grammar.
To compute $\textsc{Follow}(A)$ for all $A \in N$, apply the following rules until nothing can be added to any Follow set.

Computing Follow

Compute $\textsc{Follow}(A)$

Let $G = (N, \Sigma, S, P)$ be a grammar.
To compute $\textsc{Follow}(A)$ for all $A \in N$, apply the following rules until nothing can be added to any Follow set.

1. $\textsc{Follow}(S) := \textsc{Follow}(S) \cup \{\$\}$, where \$ is the input right endmarker.

Computing Follow

Compute $\text{Follow}(A)$

Let $G = (N, \Sigma, S, P)$ be a grammar.
To compute $\text{Follow}(A)$ for all $A \in N$, apply the following rules until nothing can be added to any Follow set.

1. $\text{Follow}(S) := \text{Follow}(S) \cup \{\$\}$, where \$ is the input right endmarker.
2. If there is a production $A \to \alpha B \beta$, then $\text{Follow}(B) := \text{Follow}(B) \cup \text{First}(\beta) - \{\varepsilon\}$.

Computing Follow

Compute $\text{Follow}(A)$

Let $G = (N, \Sigma, S, P)$ be a grammar.
To compute $\text{Follow}(A)$ for all $A \in N$, apply the following rules until nothing can be added to any Follow set.

1. $\text{Follow}(S) := \text{Follow}(S) \cup \{\$\}$, where \$ is the input right endmarker.
2. If there is a production $A \to \alpha B \beta$, then $\text{Follow}(B) := \text{Follow}(B) \cup \text{First}(\beta) - \{\varepsilon\}$.
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$ with $\varepsilon \in \text{First}(\beta)$, then $\text{Follow}(B) := \text{Follow}(B) \cup \text{Follow}(A)$.

Computing Follow (I)

1. $\textsc{Follow}(S) := \textsc{Follow}(S) \cup \{\$\}$.

2. If $A \to \alpha B \beta$, then $\textsc{Follow}(B) := \textsc{Follow}(B) \cup (\textsc{First}(\beta) - \{\varepsilon\})$.

3. If $A \to \alpha B$, or $A \to \alpha B \beta$ with $\varepsilon \in \textsc{First}(\beta)$, then
   $\textsc{Follow}(B) := \textsc{Follow}(B) \cup \textsc{Follow}(A)$.

Computing Follow (II)

1. $\textsc{Follow}(S) := \{\$\}$.
2. For every production $A \to \alpha B \beta$, then $\textsc{First}(\beta) - \{\varepsilon\} \subseteq \textsc{Follow}(B)$.
3. For every production $A \to \alpha B$, or $A \to \alpha B \beta$ with $\varepsilon \in \textsc{First}(\beta)$, then $\textsc{Follow}(A) \subseteq \textsc{Follow}(B)$.

Exercise

First and Follow

Compute the sets First and Follow for all nonteminal symbols of the following grammar:

$$S \rightarrow AB \qquad A \rightarrow aA \mid a \qquad B \rightarrow bBc \mid bc$$

Table of Contents

Motivation

Takeaway

- Top-Down parsing can be viewed as the problem of constructing a parse tree for a given input string.
- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

Predictive Parsing

- Predictive parsing chooses the correct production to use by looking ahead at the input a fixed number of symbols, typically we may look only at one (i.e., the next input symbol).

Predictive Parsing

- Predictive parsing chooses the correct production to use by looking ahead at the input a fixed number of symbols, typically we may look only at one (i.e., the next input symbol).
- This technique does not require backtracking.

Predictive Parsing

- Predictive parsing chooses the correct production to use by looking ahead at the input a fixed number of symbols, typically we may look only at one (i.e., the next input symbol).
- This technique does not require backtracking.
- This kind of parsers can be constructed for a class of grammars called LL(1).

Predictive Parsing

- Predictive parsing chooses the correct production to use by looking ahead at the input a fixed number of symbols, typically we may look only at one (i.e., the next input symbol).
- This technique does not require backtracking.
- This kind of parsers can be constructed for a class of grammars called LL(1).
- The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of look ahead at each step to make parsing action decisions.

LL(1) Grammars

Definition (LL(1) Grammars)

A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ with $\alpha \neq \beta$, the following conditions hold:

1. Neither α nor β derive strings beginning with the same terminal symbol.

LL(1) Grammars

Definition (LL(1) Grammars)

A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ with $\alpha \neq \beta$, the following conditions hold:

1. Neither α nor β derive strings beginning with the same terminal symbol.
2. At most one of α and β can derive the empty string.

LL(1) Grammars

Definition (LL(1) Grammars)

A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ with $\alpha \neq \beta$, the following conditions hold:

1. Neither α nor β derive strings beginning with the same terminal symbol.
2. At most one of α and β can derive the empty string.
3. If $\beta \xrightarrow{*} \varepsilon$, then α does not derive any string beginning with a terminal in $\textsc{Follow}(A)$.

LL(1) Grammars

Definition (LL(1) Grammars)

A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ with $\alpha \neq \beta$, the following conditions hold:

1. Neither α nor β derive strings beginning with the same terminal symbol.
2. At most one of α and β can derive the empty string.
3. If $\beta \xrightarrow{*} \varepsilon$, then α does not derive any string beginning with a terminal in $\textsc{Follow}(A)$. *Analogously*, if $\alpha \xrightarrow{*} \varepsilon$, then β does not derive any string beginning with a terminal in $\textsc{Follow}(A)$.

LL(1) Grammars

The conditions in Definition 3 can be specified as:

Whenever $A \to \alpha \mid \beta$ with $\alpha \neq \beta$:

1. $\textsc{First}(\alpha) \cap \textsc{First}(\beta) = \emptyset$.
2. If $\varepsilon \in \textsc{First}(\beta)$, then $\textsc{First}(\alpha) \cap \textsc{Follow}(A) = \emptyset$.
   If $\varepsilon \in \textsc{First}(\alpha)$, then $\textsc{First}(\beta) \cap \textsc{Follow}(A) = \emptyset$.

Exercise

LL(1)

Verify that the following grammar is LL(1):

$$S \to AB \qquad A \to aA \mid a \qquad B \to bBc \mid bc$$

Observation

Observation

- The class of LL(1) grammars is rich enough to cover most programming constructs, although care is needed in writing a suitable grammar for the source language.

Observation

Observation

- The class of LL(1) grammars is rich enough to cover most programming constructs, although care is needed in writing a suitable grammar for the source language.
- For instance, no *left-recursive* or *ambiguous* grammar can be LL(1).

Predictive Parsing Table

Let $G = (N, \Sigma, P, S)$ be a CFG. We shall construct a table $M \subseteq (N \times \Sigma \cup \{\$\}) \times \mathscr{P}(P)$.

Algorithm Construction of a predictive parsing table M

for each production $A \to \alpha \in P$ **do**
  for every $a \in \textsc{First}(\alpha)$ **do**
    $M[A,a] \leftarrow M[A,a] \cup \{A \to \alpha\}$
  if $\varepsilon \in \textsc{First}(\alpha)$ **then**
    for each terminal $b \in \textsc{Follow}(A)$ **do**
      $M[A,b] \leftarrow M[A,b] \cup \{A \to \alpha\}$
  if $\varepsilon \in \textsc{First}(\alpha) \land \$ \in \textsc{Follow}(A)$ **then**
    $M[A,\$] \leftarrow M[A,\$] \cup \{A \to \alpha\}$

If after performing the algorithm there is no production at all in some $M[A,a]$, then set $M[A,a]$ to `error` (normally represented by an empty entry in the table).

Predictive Parsing Table

Observation

- The algorithm can be applied to any grammar G to produce a parsing table M.
- However, for every LL(1) grammar, each parsing-table entry uniquely identifies a production or signals an error.
- For some grammars, M may have some entries that are multiply defined.

Exercise

Exercise

Construct the parsing table M for the grammar

$$S \to AB \qquad A \to aA \mid a \qquad B \to bBc \mid bc$$

Predictive Parsing Algorithm

Algorithm Predictive Parsing

Let T be a stack ▷ At the beginning it contains $S\$$
Let w be a string and a be its first symbol
Let X be the top stack symbol
while $X \neq \$$ **do**
    if $X = a$ **then**
        T.`pop()`
        Let a be the next symbol of w
    else if X is a terminal **then** ERROR()
    else if $M[X, a]$ is an `error` entry **then** ERROR()
    else if $M[X, a] = X \to Y_1 Y_2 \cdots Y_k$ **then**
        T.`pop()`
        T.`push`$(Y_1 Y_2 \cdots Y_k)$ ▷ Y_1 on top

Table of Contents

Motivation

Takeaway

Bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Handle Pruning

Definition (Handle-Pruning)

Let G be a grammar. If $S \xrightarrow{*} \alpha A w \rightarrow \alpha\beta w$, we say that a production $A \rightarrow \beta$ is a **handle** of $\alpha\beta w$.

Handle Pruning

Definition (Handle-Pruning)

Let G be a grammar. If $S \xrightarrow{*} \alpha A w \rightarrow \alpha \beta w$, we say that a production $A \rightarrow \beta$ is a **handle** of $\alpha \beta w$.

Objective

By "handle pruning" we can obtain a rightmost derivation.

$$S = \gamma_0 \rightarrow_{rm} \gamma_1 \rightarrow_{rm} \cdots \rightarrow_{rm} \gamma_{n-1} \rightarrow_{rm} \gamma_n = w$$

for every γ_i we have a **handle** $A_i \rightarrow \beta_i$

Steps

1. **Shift**. Shift the next input symbol onto the top of the stack.

Steps

1. **Shift**. Shift the next input symbol onto the top of the stack.
2. **Reduce**. The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.

Steps

1. **Shift**. Shift the next input symbol onto the top of the stack.
2. **Reduce**. The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. **Accept**. Announce successful completion of parsing.

Steps

1. **Shift**. Shift the next input symbol onto the top of the stack.
2. **Reduce**. The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. **Accept**. Announce successful completion of parsing.
4. **Error**. Discover a syntax error and call an error recovery routine.

Motivation

LR(k) Parsers

- The “L” is for left-to-right scanning of the input, the “R” for constructing a rightmost derivation (in reverse), and the “k” for the number of input symbols of lookahead that are used in making parsing decisions.
- LR parsers are table-driven. If we can construct a parsing table for a grammar, it is said to be and *LR grammar*.

Model

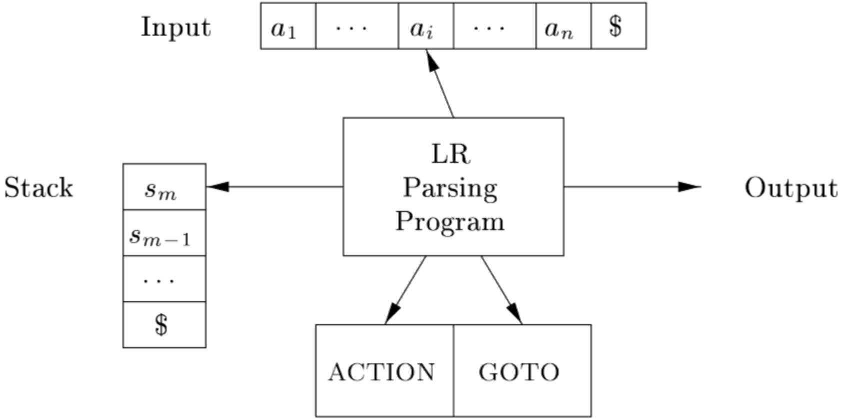


Figure 4.35 Aho et al.

LR(0) Automaton

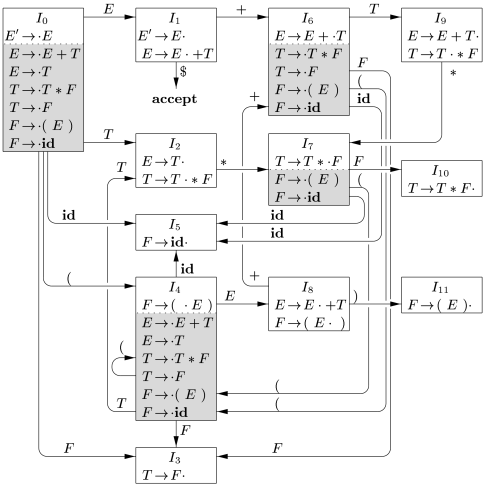


Figure 4.31 Aho et al.

Preliminaries

We need to define:

Preliminaries

- Items and closure of item sets
- Function GoTo
- Action
- Construction of parsing table
- Behavior of LR(0) automaton

Items

Definition (Items)

- Given a production $A \to \alpha$ we define an **item** to be this production with a dot "$\bullet$" somewhere in α, even at the beginning or the end.

Items

Definition (Items)

- Given a production $A \to \alpha$ we define an **item** to be this production with a dot "$\bullet$" somewhere in α, even at the beginning or the end.
- If $A \to \varepsilon$, it produces the item $A \to \bullet$.

Items

Definition (Items)

- Given a production $A \to \alpha$ we define an **item** to be this production with a dot "$\bullet$" somewhere in α, even at the beginning or the end.
- If $A \to \varepsilon$, it produces the item $A \to \bullet$.
- The set of items for a grammar G is the set of all items that can be constructed for the productions of G.

Preliminaries

Definition (Canonical LR(0))

It is the collection of sets of items that provides the basis to construct the parser.

Preliminaries

Definition (Canonical LR(0))

It is the collection of sets of items that provides the basis to construct the parser.

Definition (Augmented Grammar)

From now on, given a grammar G we work with the **augmented grammar** for G, denoted G'. It is G with a new start symbol S' and a new production $S' \to S$.

Closure of Item Sets

Definition (Closure)

If I is an item set for a grammar G, then $\textsc{Closure}(I)$ is the set of items constructed from I by the following rules:

Closure of Item Sets

Definition (Closure)

If I is an item set for a grammar G, then $\textsc{Closure}(I)$ is the set of items constructed from I by the following rules:

1. $I \subseteq \textsc{Closure}(I)$

Closure of Item Sets

Definition (Closure)

If I is an item set for a grammar G, then $\textsc{Closure}(I)$ is the set of items constructed from I by the following rules:

1. $I \subseteq \textsc{Closure}(I)$
2. If $A \rightarrow \alpha \bullet B\beta \in \textsc{Closure}(I)$ and $B \rightarrow \gamma$ is a production, then add $B \rightarrow \bullet\gamma$ to $\textsc{Closure}(I)$, if it is not already there.

Closure of Item Sets

Definition (Closure)

If I is an item set for a grammar G, then $\textsc{Closure}(I)$ is the set of items constructed from I by the following rules:

1. $I \subseteq \textsc{Closure}(I)$
2. If $A \to \alpha \bullet B \beta \in \textsc{Closure}(I)$ and $B \to \gamma$ is a production, then add $B \to \bullet \gamma$ to $\textsc{Closure}(I)$, if it is not already there.
   Apply this rule until no more new items can be added to $\textsc{Closure}(I)$.

Closure of Item Sets

Observation

There are two classes of items:

- *Kernel items*: the initial item, $S' \to \bullet S$, and all items whose dots are not at the left end.
- *Nonkernel items*: all items with their dots at the left end, except for $S' \to \bullet S$

Computing a Closure Set

Let $G = (N, \Sigma, S, P)$ be a grammar and I a set of items.

Algorithm $\textsc{Closure}(I)$

```
J ← I
repeat
    for each item A → α•Bβ ∈ J do
        for each production B → γ ∈ P do
            if B → •γ ∉ J then
                J ← J ∪ {B → •γ}
until no more items are added to J
```

The Function GoTo

Definition (GoTo)

Let I be a set of items and X a grammar symbol. $\textsc{GoTo}(I, X)$ is defined to be the closure of the set of all items $A \to \alpha X \bullet \beta$ such that $A \to \alpha \bullet X \beta \in I$.

States of the LR(0) Automaton

Given an augmented grammar G' we are going to construct the LR(O) automaton by using $\textsc{Closure}(\_)$ and $\textsc{GoTo}(\_,\_)$.

States of the LR(0) Automaton

Given an augmented grammar G' we are going to construct the LR(O) automaton by using $\textsc{Closure}(\_)$ and $\textsc{GoTo}(\_,\_)$.

- The initial state, I_0, of the parser is the closure of the set of items containing $S' \rightarrow \bullet S$.

States of the LR(0) Automaton

Given an augmented grammar G' we are going to construct the LR(O) automaton by using $\textsc{Closure}(\_)$ and $\textsc{GoTo}(\_,\_)$.

- The initial state, I_0, of the parser is the closure of the set of items containing $S' \to \bullet S$.
- By using $\textsc{GoTo}(\_,\_)$, we create both the other states and the transitions of the automaton.

Constructing Canonical LR(0)

Algorithm Computes the set of items of an extended grammar G'

procedure $\textsc{SetsOfItems}(G')$
  $C \leftarrow \{\textsc{Closure}(\{S' \rightarrow \bullet S\})\}$
  repeat
    for each set of items $I \in C$ **do**
      for each grammar symbol X **do**
        if $\textsc{GoTo}(I, X) \neq \emptyset \wedge \textsc{GoTo}(I, X) \notin C$ **then**
          $C \leftarrow C \cup \{\textsc{GoTo}(I, X)\}$
  until no more sets of items are added to C

The Function Action

Definition (Action)

Let i be a state (of the parser we are constructing) and a be a terminal (or \$). The value of $\textsc{Action}(i, a)$ is given by:

The Function Action

Definition (Action)

Let i be a state (of the parser we are constructing) and a be a terminal (or \$). The value of $\textsc{Action}(i, a)$ is given by:

1. **Shift** j, where j is a state. This shifts input a to the stack, a is represented by j.

The Function Action

Definition (Action)

Let i be a state (of the parser we are constructing) and a be a terminal (or \$). The value of $\textsc{Action}(i,a)$ is given by:

1. **Shift** j, where j is a state. This shifts input a to the stack, a is represented by j.
2. **Reduce** $A \to \beta$. Reduces β on the top of the stack to head A.

The Function Action

Definition (Action)

Let i be a state (of the parser we are constructing) and a be a terminal (or \$). The value of $\textsc{Action}(i,a)$ is given by:

1. **Shift** j, where j is a state. This shifts input a to the stack, a is represented by j.
2. **Reduce** $A \to \beta$. Reduces β on the top of the stack to head A.
3. **Accept**. It accepts the input and finishes parsing.

The Function Action

Definition (Action)

Let i be a state (of the parser we are constructing) and a be a terminal (or \$). The value of $\textsc{Action}(i,a)$ is given by:

1. **Shift** j, where j is a state. This shifts input a to the stack, a is represented by j.
2. **Reduce** $A \to \beta$. Reduces β on the top of the stack to head A.
3. **Accept**. It accepts the input and finishes parsing.
4. **Error**. The parser discovers an error in its input and takes some corrective action.

Use of the LR(0) Automaton

SLR

We now construct a *simple LR* (SLR) parser based on the LR(0) automaton.
We begin by constructing the SLR-parsing table.

Constructing SLR-Parsing Tables

Constructing an SLR-parsing table

Let G' be an augmented grammar.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the canonical LR(0)

Constructing SLR-Parsing Tables

Constructing an SLR-parsing table

Let G' be an augmented grammar.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the canonical LR(0)
2. State i is constructed from I_i.

Constructing SLR-Parsing Tables

Constructing an SLR-parsing table

Let G' be an augmented grammar.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the canonical LR(0)
2. State i is constructed from I_i.
   2.1 If $A \rightarrow \alpha \bullet a \beta \in I_i$, with $a \in \Sigma$ and $\textsc{GoTo}(I_i, a) = I_j$, then $\textsc{Action}(i, a) \leftarrow$ "shift j"

Constructing SLR-Parsing Tables

Constructing an SLR-parsing table

Let G' be an augmented grammar.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the canonical LR(0)
2. State i is constructed from I_i.
   2.1 If $A \to \alpha \bullet a\beta \in I_i$, with $a \in \Sigma$ and $\textsc{GoTo}(I_i, a) = I_j$, then $\textsc{Action}(i, a) \leftarrow$ "shift j"
   2.2 If $A \to \alpha \bullet \in I_i$ with $A \neq S'$, then $\textsc{Action}(i, a) \leftarrow$ "reduce $A \to \alpha$" for all $a \in \textsc{Follow}(A)$

Constructing SLR-Parsing Tables

Constructing an SLR-parsing table

Let G' be an augmented grammar.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the canonical LR(0)
2. State i is constructed from I_i.
   2.1 If $A \to \alpha \bullet a\beta \in I_i$, with $a \in \Sigma$ and $\textsc{GoTo}(I_i, a) = I_j$, then $\textsc{Action}(i, a) \leftarrow$ "shift j"
   2.2 If $A \to \alpha \bullet \in I_i$ with $A \neq S'$, then $\textsc{Action}(i, a) \leftarrow$ "reduce $A \to \alpha$" for all $a \in \textsc{Follow}(A)$
   2.3 If $S' \to S \bullet \in I_i$, then $\textsc{Action}(i, \$) \leftarrow$ "accept"

Constructing SLR-Parsing Tables

Constructing an SLR-parsing table

Let G' be an augmented grammar.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the canonical LR(0)
2. State i is constructed from I_i.
   2.1 If $A \to \alpha \bullet a\beta \in I_i$, with $a \in \Sigma$ and $\textsc{GoTo}(I_i, a) = I_j$, then $\textsc{Action}(i, a) \leftarrow$ "shift j"
   2.2 If $A \to \alpha \bullet \in I_i$ with $A \neq S'$, then $\textsc{Action}(i, a) \leftarrow$ "reduce $A \to \alpha$" for all $a \in \textsc{Follow}(A)$
   2.3 If $S' \to S\bullet \in I_i$, then $\textsc{Action}(i, \$) \leftarrow$ "accept"

   If any conflicting actions result from the above rules, we say the grammar is not SLR(1)

Constructing SLR-Parsing Tables

Constructing an SLR-parsing table

Let G' be an augmented grammar.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the canonical LR(0)
2. State i is constructed from I_i.
   2.1 If $A \to \alpha \bullet a\beta \in I_i$, with $a \in \Sigma$ and $\textsc{GoTo}(I_i, a) = I_j$, then $\textsc{Action}(i, a) \leftarrow$ "shift j"
   2.2 If $A \to \alpha \bullet \in I_i$ with $A \neq S'$, then $\textsc{Action}(i, a) \leftarrow$ "reduce $A \to \alpha$" for all $a \in \textsc{Follow}(A)$
   2.3 If $S' \to S \bullet \in I_i$, then $\textsc{Action}(i, \$) \leftarrow$ "accept"

   If any conflicting actions result from the above rules, we say the grammar is not SLR(1)
3. If $\textsc{GoTo}(I_i, A) = I_j$, then $\textsc{GoTo}(i, A) \leftarrow j$

Constructing SLR-Parsing Tables

Constructing an SLR-parsing table

Let G' be an augmented grammar.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the canonical LR(0)
2. State i is constructed from I_i.
   - 2.1 If $A \rightarrow \alpha \bullet a \beta \in I_i$, with $a \in \Sigma$ and $\textsc{GoTo}(I_i, a) = I_j$, then $\textsc{Action}(i, a) \leftarrow$ "shift j"
   - 2.2 If $A \rightarrow \alpha \bullet \in I_i$ with $A \neq S'$, then $\textsc{Action}(i, a) \leftarrow$ "reduce $A \rightarrow \alpha$" for all $a \in \textsc{Follow}(A)$
   - 2.3 If $S' \rightarrow S \bullet \in I_i$, then $\textsc{Action}(i, \$) \leftarrow$ "accept"

   If any conflicting actions result from the above rules, we say the grammar is not SLR(1)
3. If $\textsc{GoTo}(I_i, A) = I_j$, then $\textsc{GoTo}(i, A) \leftarrow j$

All entries not defined by (2) and (3) are made "error"

LR-parsing Algoritm

Let G be a grammar and w an input string.

Algorithm LR-parsing. Returns a reduction for w, otherwise an error.

```
Let T be a stack                                  ▷ At the beginning T contains 0$
Let a be the first symbol of w$
while true do
    Let s be the state on top of the stack
    if ACTION(s,a) = shift t then
        T.push(t)
        Let a be the next input symbol
    else if ACTION(s,a) = reduce A → β then
        pop |β| symbols of the stack
        T.push(GOTO(t,A))                         ▷ t is the temporary top of the stack
        output the production A → β
    else if ACTION(s,a) = accept then break       ▷ Parsing is done
    else error
```

Example

Example

Construct an LR-parser for the grammar

1. $E \to E + T$	3. $T \to T * F$	5. $F \to (E)$
2. $E \to T$	4. $T \to F$	6. $F \to \mathbf{i}$

Example

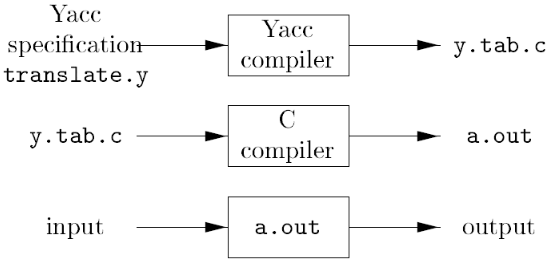
STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Table 4.37, Aho et al.

Table of Contents

Parser Generators

`Yacc`



- Declarations
- Translation rules
- Supporting C routines

Example: Grammar Specification

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \textbf{digit}$$

```
line    : expr '\n'          { printf("%d\n", $1); }
        ;
expr    : expr '+' term      { $$ = $1 + $3; }
        | term
        ;
term    : term '*' factor    { $$ = $1 * $3; }
        | factor
        ;
factor  : '(' expr ')'       { $$ = $2; }
        | DIGIT
        ;
```

Example: Complete Specification

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line   : expr '\n'        { printf("%d\n", $1); }
       ;
expr   : expr '+' term    { $$ = $1 + $3; }
       | term
       ;
term   : term '*' factor  { $$ = $1 * $3; }
       | factor
       ;
factor : '(' expr ')'     { $$ = $2; }
       | DIGIT
       ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

Example: Grammar Specification

$$
\begin{aligned}
E \to\ & E + E \mid E - E \mid E * E \\
& \mid E/E \mid -E \mid (E) \mid \textbf{number}
\end{aligned}
$$

```
lines : lines expr '\n'   { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
expr  : expr '+' expr     { $$ = $1 + $3; }
      | expr '-' expr     { $$ = $1 - $3; }
      | expr '*' expr     { $$ = $1 * $3; }
      | expr '/' expr     { $$ = $1 / $3; }
      | '(' expr ')'      { $$ = $2; }
      | '-' expr  %prec UMINUS { $$ = - $2; }
      | NUMBER
      ;
```

Example: Complete Specification

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double  /* double type for Yacc stack */
%}
%token NUMBER

%left '+' '-'
%left '*' '/'
%right UMINUS
%%

lines : lines expr '\n'   { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
expr  : expr '+' expr     { $$ = $1 + $3; }
      | expr '-' expr     { $$ = $1 - $3; }
      | expr '*' expr     { $$ = $1 * $3; }
      | expr '/' expr     { $$ = $1 / $3; }
      | '(' expr ')'      { $$ = $2; }
      | '-' expr  %prec UMINUS { $$ = - $2; }
      | NUMBER
      ;
%%
yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c == '.') || (isdigit(c)) ) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

Referencias

Aho, Alfred V. et al. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0321486811.

Kozen, Dexter C. (1997). *Automata and Computability*. en. New York, NY: Springer New York. ISBN: 978-1-4612-7309-7 978-1-4612-1844-9. DOI: 10.1007/978-1-4612-1844-9. URL: http://link.springer.com/10.1007/978-1-4612-1844-9 (visited on 01/23/2024).

Milner, R. (1999). *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, Cambridge, UK.

Reghizzi, Stefano Crespi, Luca Breveglieri, and Angelo Morzenti (2019). *Formal Languages and Compilation*. 3rd. Springer Publishing Company, Incorporated. ISBN: 3030048780.