Lenguajes Formales y Compiladores

Syntax Analysis

Sergio Ramírez Rico

UNIVERSIDAD EAFIT

Área Ciencias Fundamentales
Escuela de Ciencias Aplicadas e Ingeniería

Contenido

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 $\textsc{Recursive-parser}(A)$
    Choose an A-production: $A \longrightarrow x_1 x_2 \cdots x_k$ $\triangleright\ x_i \in N \cup \Sigma$
    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

$$
\begin{aligned}
S &\longrightarrow cAd \\
A &\longrightarrow ab \mid a
\end{aligned}
$$

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

Exercise I

Exercise

Implement a recursive parser based on Procedure 1.

Input: Assume that the input is given as follows:

1. First, three positive numbers n, m, k where n is the number of nonterminals in the grammar, m is the number of grammar's rules, and k is the number of strings to be analysed.
2. Then, a single line with the n nonterminals separated by blank spaces.
3. Then, the m rules of the grammar, a single rule for each line. A rule $A \longrightarrow \alpha$ is given as $\mathtt{A}{-}\alpha$.
4. Finally, the k strings to be analised, a single rule for each line.

Exercise II

Output: For each string, print `yes` if the grammar generates it and `no` if it does not.

Input example

Input	**Output**
`1 2 3`	`yes`
`S`	`yes`
`S-aSb`	`no`
`S-c`	
`aacbb`	
`acb`	
`ab`	

Exercise III

Input example

Input	Output
`2 4 4`	`yes`
`S A`	`no`
`S-aSb`	`yes`
`S-A`	`no`
`A-aA`	
`A-a`	
`aaabb`	
`aabb`	
`aaaaaaaaabbbb`	
`ab`	

Contenido

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.

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 $x \longrightarrow y_1 y_2 \cdots y_k$ is a production for some $k \geq 1$, then:
   - 2.1 Place a in $\textsc{First}(x)$ if for some $1 \leq i \leq k$, $a \in \textsc{First}(y_i)$, and $\varepsilon \in \bigcap_{j=1}^{i-1} \textsc{First}(y_j)$.
   - 2.2 If $\varepsilon \in \textsc{First}(y_j)$ for all $1 \leq j \leq k$, then add ε to $\textsc{First}(x)$.
3. If $x \longrightarrow \varepsilon$ is a production, then add ε to $\textsc{First}(x)$.

First

Compute $\textsc{First}(x)$

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

First

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 $1 \leq i \leq n$, $\varepsilon \in \textsc{First}(x_i)$.

This can be formalized as:

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

Follow

Definition (Follow)

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

Follow

Definition (Follow)

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

Compute $\textsc{Follow}(A)$

To compute $\textsc{Follow}(A)$ for all $A \in N$, apply the following rules until nothing can be added to any Follow set.

1. Place \$ in $\textsc{Follow}(S)$ where S is the start symbol and \$ is the input right endmarker.
2. If there is a production $A \longrightarrow \alpha B \beta$, then add $\textsc{First}(\beta) \setminus \{\varepsilon\}$ to $\textsc{Follow}(B)$.
3. If there is a production $A \longrightarrow \alpha B$, or a production $A \longrightarrow \alpha B \beta$ with $\varepsilon \in \textsc{First}(\beta)$, then add $\textsc{Follow}(A)$ to $\textsc{Follow}(B)$.

Follow

Let $G = (N, \Sigma, P, S)$ be a grammar.

Compute $\textsc{Follow}(A)$

To compute $\textsc{Follow}(A)$ for all nonterminals A, apply the following rules until nothing can be added to any Follow set.

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

Exercise

First and Follow

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

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

Contenido

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

Takeaway

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.

LL(1) Grammars

Takeaway

- Predictive parsers (recursive-descent parsers needing no backtracking), 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 $A \longrightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
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

Definition (LL(1) Grammars)

A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
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 $\textsc{Follow}(A)$.

The above conditions can be simplified as:

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):

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

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 are going to construct a two-dimensional array $M \subseteq N \times \Sigma \cup \{\$\} \times \mathscr{P}(P)$.

Construction of a predictive parsing table.

INPUT: Grammar G.
OUTPUT: Parsing table M.
For each production $A \longrightarrow \alpha$ of the grammar do as follows:

1. For each terminal $a \in \textsc{First}(\alpha)$, add $A \longrightarrow \alpha$ to $M[A, a]$.
2. If $\varepsilon \in \textsc{First}(\alpha)$, then for each terminal $b \in \textsc{Follow}(A)$, add $A \longrightarrow \alpha$ to $M[A, b]$.
3. If $\varepsilon \in \textsc{First}(\alpha)$ and $\$ \in \textsc{Follow}(A)$, add $A \longrightarrow \alpha$ to $M[A, \$]$ as well.

If after performing the above 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

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

Predictive Parsing Algorithm

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**
    Pop the stack
    Let a be the next sympol of w
  else if X is a terminal **then** $\text{ERROR}()$
  else if $M[X, a]$ is an error entry **then** $\text{ERROR}()$
  else if $M[X, a] = X \longrightarrow Y_1 Y_2 \cdots Y_k$ **then**
    pop the stack
    push $Y_1 Y_2 \cdots Y_k$ $\triangleright$ Y_1 on top

Contenido

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 $S \xrightarrow{*} \alpha A w \longrightarrow \alpha\beta w$, we say that a production $A \longrightarrow \beta$ is a handle of $\alpha\beta w$.

Handle Pruning

Definition (Handle-Pruning)

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

Objective

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

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

for every γ_i we have a handle $A_i \longrightarrow \beta_i$

Steps

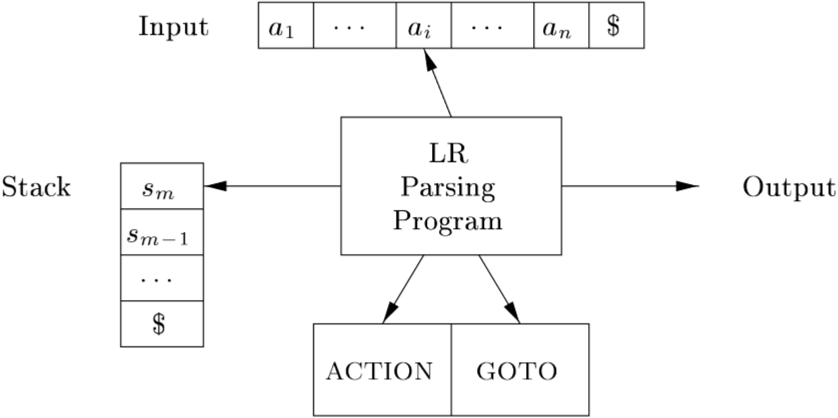
1. *Shift*. Shift the next input symbol onto the top of the stack.
2. *Reduce*. The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. *Accept*. Announce successful completion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


Input
a_1
$\cdots$
a_i
$\cdots$
a_n
\$
Stack
s_m
s_{m-1}
$\cdots$
\$
LR
Parsing
Program
Output
ACTION
GOTO

Referencias

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

[2] Dexter C. Kozen. *Automata and Computability*. 1st. Berlin, Heidelberg: Springer-Verlag, 1997. ISBN: 0387949070.

[3] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, Cambridge, UK, 1999.

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