The role of the parsing process is to reconstruct a derivation of a given input string.

In the context-free grammar, each non-terminal generates a tree fragment. Each recursive descent procedure recognizes an instance of a non-terminal and accepts the corresponding part of the input. Each terminal corresponds to one recursive descent procedure, which translates the corresponding part of the input into a call to the lexical scanner. Each recognizing function returns a tree fragment.

General Structure

Each right-hand side of a production provides a body for a function. Each non-terminal on the right is translated into a call to the function that recognizes that non-terminal. Each terminal on the right is translated into a call to the lexical scanner. Error if the resulting token is not the expected terminal.

Recur-sive Descent Parsing

Top-down parsing builds a tree from root symbol. Derivation process, which attempts to emulate the recognition process, constructs the parsing tree which represents a given input string. Equivalently, construct the parsing tree which represents a derivation of the given input string.

Each terminal in the rhs is translated into a call to the lexical scanner. Error if the resulting token is not the expected terminal. Each non-terminal in the rhs is translated into a call to the procedure that recognizes that non-terminal. Each production corresponds to one recursive descent procedure.

The parsing process

A.P. Pnueli

Honors Compilers, NYU, Spring, 2007

Lecture 3: Parsing

A. Pnueli

Honors Compilers, NYU, Spring, 2007

Lecture 3: Parsing

A. Pnueli

Honors Compilers, NYU, Spring, 2007

Lecture 3: Parsing

A. Pnueli

Honors Compilers, NYU, Spring, 2007
Example: Parsing a Declaration

FULL.TYPE.DECLARATION $::=
  type DEFINING.DEFINITION is TYPE.DEFINITION

Translates into:
- Find a defining-identifier
- Recognize a type-definition
- Function call for get token

In practice, we already know that the first token is "type".
This is why this procedure was called in the first place.

Complications

If there are multiple productions for a non-terminal, we need a mechanism to determine which production to use.

Example: Parsing a Loop

FOR-STATEMENT $::=
  ITERATION.SCHEME loop STATEMENTS end loop

Translates into:
- Function call for get token loop
- Function call for get token end
- Function call for semicolon

In case we fail to find any of the expected tokens or one of the called functions returns a failure, this function returns a failure indication.

Solution: Factor Grammar

IF-STAT $::=
  if COND then Stats [ELSE PART] end if;

Problem now reduces to recognizing whether an optional component (ELSE.PART) is present. We use a single token lookahead this is possible — look for a token "else".
Illustrate Solution

Consider rule

Boolean function get IF ()

if Token = if then Scan else return 0;

if get COND () = 0 then return 0;

if get COND () = 1 then return 0;

if get COND () = 0 then return 0;

if get COND () = 1 then return 0;

boolean function get IF ()
Arranging the non-terminals in some order $A_1, \ldots, A_n$.
A context-free language can be recognized by a PDA, which is a type of automaton with a stack. The PDA is defined by a tuple $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where:

- $Q$ is a finite set of states.
- $\Sigma$ is the input alphabet.
- $\Gamma$ is the stack alphabet.
- $\delta$ is a transition function.
- $q_0 \in Q$ is the initial state.
- $Z_0 \in \Gamma$ is the initial stack symbol.
- $F \subseteq Q$ is the set of accepting states.

The transition function $\delta$ maps a state $q$, an input symbol $a$, and a stack symbol $Z$ to a new state $q'$ and a new stack symbol $Z'$. Formally, $\delta(q, a, Z) = (q', Z')$.

A PDA can accept a language by reaching a state in $F$ with an empty stack. The language recognized by a PDA is the set of all inputs that lead to an accepting state with an empty stack.

Formally, if $L(A)$ is the language recognized by a PDA $A$, then $L(A) = \{ w \in \Sigma^* \mid \text{some } q_0 \in Q \text{ and } Z_0 \in \Gamma \text{ lead to } q \in F \text{ with } Z = \epsilon \text{ on } \delta \text{ transitions} \}$. 

**Example:**

Consider the language of balanced parentheses expressions. A PDA that accepts this language is defined as follows:

1. **States:** $Q = \{ q_0, q_1, q_2 \}$
2. **Input alphabet:** $\Sigma = \{ (, ) \}$
3. **Stack alphabet:** $\Gamma = \{ \text{L}, \text{R} \}$
4. **Initial state:** $q_0$
5. **Initial stack symbol:** $Z_0 = \epsilon$
6. **Accepting states:** $F = \{ q_2 \}$
7. **Transition function:**
   - $\delta(q_0, \text{L}, \epsilon) = (q_1, \text{L})$
   - $\delta(q_1, \text{L}, \text{L}) = (q_1, \text{L})$
   - $\delta(q_1, \text{R}, \text{R}) = (q_1, \text{R})$
   - $\delta(q_1, \text{L}, \text{R}) = (q_2, \text{R})$
   - $\delta(q_2, \epsilon, \epsilon) = (q_2, \epsilon)$

The PDA can read the input and write to the top of the stack, and the transition table is defined by the set of internal states and a transition function. For example, the language of balanced parentheses expressions can be generated by the following grammar:

$$S \rightarrow () | (S) | SS$$
A PDA is defined to be deterministic (DPDA) if it has no \( \varepsilon \)-moves, and for every \( b \in \Gamma \), \( a \in \Sigma \), and \( X \in \Delta \),
\[
|\{ (X,a,b) | \} | < I
\]
for non-terminal \( X \), attach to parent \( T \).  

Semantic Action when choosing a production, build the node.

- If stack and input string are both empty, apply then accept.
- Otherwise, choose a grammar production \( T \rightarrow a \). Replace \( T \).
- If \( T \) is a terminal symbol, then \( T \) must equal \( a \). Pop stack and consume input. This is called a match action.
- If \( T \) is a non-terminal symbol, then \( T \) must equal \( a \). Pop stack and consume input. This is called a non-match action.
- At each step, let \( T \) be the symbol at top of the stack and \( a \) be the next input token.

Initially, stack contains Grammar start symbol \( S \).

\[
\text{Actions}
\]

- \( S \rightarrow (()) \triangleleft (s()) \triangleleft (ss) \triangleleft (s) \triangleleft S
\]

Consider the grammar:
\[
S ::= ()
| (S)
| SS
\]

and the leftmost derivation:
\[
S \Rightarrow (()) \Rightarrow ((SS)) \Rightarrow ((())S) \Rightarrow (())()
\]

A top-down recognition by a PDA is given by:
\[
\begin{array}{c|c}
\text{Stack} & \text{Input} & \text{Action} \\
(())(()) & (s()) & \text{Expand} \\
((s)) & (ss) & \text{Match} \\
(s) & (s) & \text{Expand} \\
((s)) & () & \text{Expand} \\
() & () & \text{Match} \\
() & () & \text{Match} \\
() & () & \text{Match} \\
& & \text{Accept}
\end{array}
\]
include the following rules:

\[ (S \rightarrow a \{L, S \}, 0) : \delta \]

Following the recipe of Claim 4, we obtain the grammar where the productions

\[
\begin{array}{c|cc}
\emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset \\
G & \emptyset & \emptyset \\
\end{array}
\]

Consider the PDA

\[ \begin{array}{c|cc}
\emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset \\
G & \emptyset & \emptyset \\
\end{array} \]

Example: Converting a PDA to a CFG

Claim 3. For every CFG \( G \) there exists a PDA \( A \) which accepts by

Correspondence of PDA's to CFG's

We Need Deterministic Parsing

Claim 5. Every PDA is equivalent to a single-stack PDA.

What about multi-stack PDA's?

empty stack, then \( Z \).

For every production \( \alpha \rightarrow \beta \), there exists a grammar which generates the language recognized by \( A \) with empty stack.

Proof:

For every single-stack PDA \( A \) there exists a CFG \( G \) which generates the language recognized by \( A \) with empty stack.

Correspondence of PDA's to CFG's

include the following rules:

\[ (S \rightarrow a \{L, S \}, 0) : \delta \]

Following the recipe of Claim 4, we obtain the grammar where the productions

\[
\begin{array}{c|cc}
\emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset \\
G & \emptyset & \emptyset \\
\end{array}
\]

Consider the PDA

\[ \begin{array}{c|cc}
\emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset \\
G & \emptyset & \emptyset \\
\end{array} \]

Example: Converting a PDA to a CFG

Claim 3. For every CFG \( G \) there exists a PDA \( A \) which accepts by
Computing FIRST

For a string derived from a non-terminal $X$, we say that

- If $X$ is a production, add $\epsilon$ to $\text{FIRST}(X)$.
- Add $\epsilon$ to $\text{FIRST}(X)$ for all $\alpha \in \text{FIRST}(X)$.
- For a string derived from a terminal $a$, add $a$ to $\text{FIRST}(X)$.

Also, for some $X$, $X$ is a production, and $a$ is a terminal, we say that

- If $X$ is a production, add $\epsilon$ to $\text{FIRST}(X)$.
- Add $\epsilon$ to $\text{FIRST}(X)$ for all $\alpha \in \text{FIRST}(X)$.
- Add $\epsilon$ to $\text{FIRST}(X)$ for $\alpha \in \text{FIRST}(X)$.

For each non-terminal $X$ and production

$$\{X\} = \text{FIRST}(X) \setminus \{\epsilon\}$$

Otherwise we choose $\epsilon$.

If the next input character is $\epsilon$, we choose

$$\epsilon \mid S(S) =:: S$$

The following grammar is $\text{LL}(1)$:

$$SS \mid (S) \mid (\epsilon) =:: S$$

Examples

The following grammar for balanced parentheses is not $\text{LL}(k)$ for any $k$.

$$S \rightarrow S(S) \mid (S) \mid (\epsilon)$$

Converting $\text{LL}(1)$ Tables

Let $k$ be a positive integer. The grammar $G$ is called an $\text{LL}(k)$ grammar if for every leftmost derivation $S \Rightarrow^* A \epsilon$ and the first character of $A$ is not a terminal, $A \epsilon$ is uniquely determined by $A$ and the $k$ first characters of $A$. The name is based on the fact that parsing according to such a grammar reads the input from left to right while constructing a leftmost derivation with a look-ahead of $k$ characters.

A 1-lookahead is sufficient in order to distinguish between $S(S)$ and $(S)$ (and vice versa). However, no bounded lookahead is sufficient in order to distinguish $SS \mid (S) \mid (\epsilon)$.

Examples

The following grammar for balanced parentheses is not $\text{LL}(k)$ for any $k$.

$$S \rightarrow S(S) \mid (S)$$

After a string derived from $A$.

$$\text{FIRST}(A) \subseteq \{\epsilon\} \cup \{ax \mid a \in \{\text{FIRST}(A)\}, x \in \epsilon\}$$

$\text{FOLLOW}(A) = \text{FIRST}(\epsilon)$ if the set of terminal symbols that can appear after a string derived from $A$.

$$\{\epsilon \in \{\text{FIRST}(A)\} \cup \{ax \mid a \in \{\text{FIRST}(A)\}, x \in \epsilon\} = \text{FIRST}(A)$$

For a non-terminal $A \in \text{FIRST}(A)$, $\text{FOLLOW}(A)$ is the set of terminal symbols that can appear after the first character in a string derived from $A$.

Define two functions on the symbols of the grammar:

$\text{FIRST}$ and $\text{FOLLOW}$
Constructing an $LL(1)$ Parsing Table

This is a table $M[A,a]$, which, for each non-terminal production $A \rightarrow a$ of the grammar, do the following:

- If there is a production $A \rightarrow \alpha B \beta$, then add all symbols in $\text{FIRST}(B) - \{\epsilon\}$ to $\text{FOLLOW}(B)$.
- If there is a production $A \rightarrow \alpha B$, then add all symbols in $\text{FIRST}(A)$ to $\text{FOLLOW}(B)$.
- If $\epsilon \in \text{FIRST}(\alpha)$ then, for each terminal $b \in \text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A,b]$. If $\epsilon \in \text{FIRST}(\alpha)$ and $\epsilon \in \text{FOLLOW}(A)$, then add $A \rightarrow \alpha$ to $M[A,\epsilon]$ as well.

We can parse with the following input:

```
E → +TE  T → FT  T → *FT  F → (E)  F → id  +  E  T  F  $  id  +  T  F  E
```

This is a table $\text{FOLLOW}(X)$, where $X$ is the input end-marker, $S$ is the start symbol, and $E$ is the input start symbol.

### Example: Constructing $LL(1)$ Tables

We construct the $\text{FIRST}$ tables:

- $E$: $\{id\}$
- $F$: $\{id\}$
- $T$: $\{id, +, \epsilon\}$
- $S$: $\{id\}$

Leading to the following parsing table:

```
<table>
<thead>
<tr>
<th>Action</th>
<th>Input</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow +TE'$</td>
<td>$E \rightarrow +TE'$</td>
<td>$E \rightarrow +TE'$</td>
</tr>
<tr>
<td>$T \rightarrow FT'$</td>
<td>$T \rightarrow FT'$</td>
<td>$T \rightarrow FT'$</td>
</tr>
<tr>
<td>$T \rightarrow *FT'$</td>
<td>$T \rightarrow *FT'$</td>
<td>$T \rightarrow *FT'$</td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>$F \rightarrow (E)$</td>
<td>$F \rightarrow (E)$</td>
</tr>
<tr>
<td>$F \rightarrow id$</td>
<td>$F \rightarrow id$</td>
<td>$F \rightarrow id$</td>
</tr>
<tr>
<td>$+ \rightarrow +$</td>
<td>$+ \rightarrow +$</td>
<td>$+ \rightarrow +$</td>
</tr>
<tr>
<td>$\epsilon \rightarrow \epsilon$</td>
<td>$\epsilon \rightarrow \epsilon$</td>
<td>$\epsilon \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$id \rightarrow id$</td>
<td>$id \rightarrow id$</td>
<td>$id \rightarrow id$</td>
</tr>
<tr>
<td>$T \rightarrow FT'$</td>
<td>$T \rightarrow FT'$</td>
<td>$T \rightarrow FT'$</td>
</tr>
<tr>
<td>$T \rightarrow *FT'$</td>
<td>$T \rightarrow *FT'$</td>
<td>$T \rightarrow *FT'$</td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>$F \rightarrow (E)$</td>
<td>$F \rightarrow (E)$</td>
</tr>
<tr>
<td>$F \rightarrow id$</td>
<td>$F \rightarrow id$</td>
<td>$F \rightarrow id$</td>
</tr>
</tbody>
</table>
```

### Computing $\text{FOLLOW}(X)$

- Place $\$$ in $\text{FOLLOW}(S)$, where $S$ is the start symbol, and $\$$ is the input start symbol.
- For each terminal $A \rightarrow \alpha$ of the grammar, do the following:
  - If there is a production $A \rightarrow \alpha B \beta$, then add all symbols in $\text{FIRST}(B) - \{\epsilon\}$ to $\text{FOLLOW}(B)$.
  - If there is a production $A \rightarrow \alpha B$, then add all symbols in $\text{FIRST}(A)$ to $\text{FOLLOW}(B)$.

This is a table $M[A,a]$, which, for each non-terminal production $A \rightarrow a$, tell us which $A \in N$ and next input character $a \in T$, do the following:

- If $\epsilon \in \text{FIRST}(\alpha)$ then, for each terminal $b \in \text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A,b]$. If $\epsilon \in \text{FIRST}(\alpha)$ and $\epsilon \in \text{FOLLOW}(A)$, then add $A \rightarrow \alpha$ to $M[A,\epsilon]$ as well.
Claim 6. A grammar $G$ is an LL(1) grammar if the parsing table $M[A,a]$ contains at most one production in each entry.

Correctness of the Construction