Front End: Syntax Analysis

The Role of the Parser

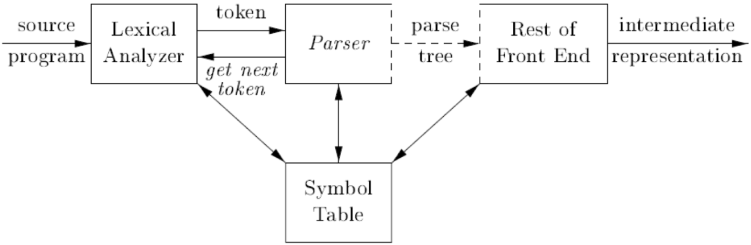


Figure 4.1: Position of parser in compiler model

The Role of the Parser

- Construct a parse tree
- Report and recover from errors
- Collect information into symbol tables

Types of Parsers

- There are three general types of parsers for grammars:
  - Universal
  - Top-down
  - Bottom-up
- In compilers, the methods commonly used are either top-down or bottom-up.
- One input symbol at a time, from left to right.
- Efficiency is achieved by restricting to particular grammars: **LL** (manually) or **LR** (automated tools).

Grammars for expressions

- Universal methods are suitable for general grammars, e.g.

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

(no associativity, no precedence captured)

- Bottom-up methods: **LR** grammars, e.g.

$$\begin{array}{rcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

(associativity and precedence captured)

- Top-down methods: **LL** grammars, e.g.

$$\begin{array}{rcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \varepsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \varepsilon \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

Context-free Grammars

A *Context-free grammar* (or *grammar*) systematically describes the syntax of programming language constructs.

$$
\begin{aligned}
expression &\rightarrow expression + term \\
expression &\rightarrow expression - term \\
expression &\rightarrow term \\
term &\rightarrow term * factor \\
term &\rightarrow term\ /\ factor \\
term &\rightarrow factor \\
factor &\rightarrow (\ expression\ ) \\
factor &\rightarrow \mathbf{id}
\end{aligned}
$$

Figure 4.2: Grammar for simple arithmetic expressions

Terminal symbols: **id** + - * / ( ) Non-terminal: *expression*, *term*, *factor*. Start symbol: *expression*

CFG: Formal Definition

$$G = (T, N, P, S)$$

- T is a finite set of terminals
- N is a finite set of non-terminals
- P is a finite subset of production rules of the form
  - $A \rightarrow \alpha_1 \alpha_2 \ldots \alpha_k$ with $A \in N$, $\alpha_i \in T \cup N$
- S is the start symbol
  - $S \in N$

Derivations

Using notational conventions the grammar in Fig.4.2 becomes

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

A derivation of a string of terminals in this grammar is a proof that the string is an expression.
Leftmost derivation: always choose the leftmost nonterminal

$$E \Rightarrow^{lm} E + T \Rightarrow^{lm} \mathbf{id} + T \Rightarrow^{lm} \mathbf{id} + F \Rightarrow^{lm} \mathbf{id} + \mathbf{id}$$

Rightmost derivation: always choose the righttmost nonterminal

$$E \Rightarrow^{rm} E{+}T \Rightarrow^{rm} E{+}F \Rightarrow^{rm} E{+}\mathbf{id} \Rightarrow^{rm} T{+}\mathbf{id} \Rightarrow^{rm} F{+}\mathbf{id} \Rightarrow^{rm} \mathbf{id}{+}\mathbf{id}$$

Parse Trees

A parse tree is a graphical representation of a derivation: an interior node represents the head of a production; its children are labelled by the symbols in the body.

$$E \quad \rightarrow \quad E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id}$$

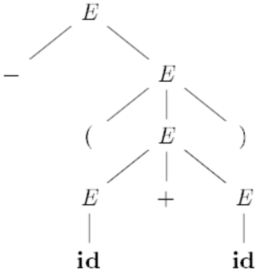


Figure 4.3: Parse tree for $-(\mathbf{id} + \mathbf{id})$

Example

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id}) \qquad (4.8)$$

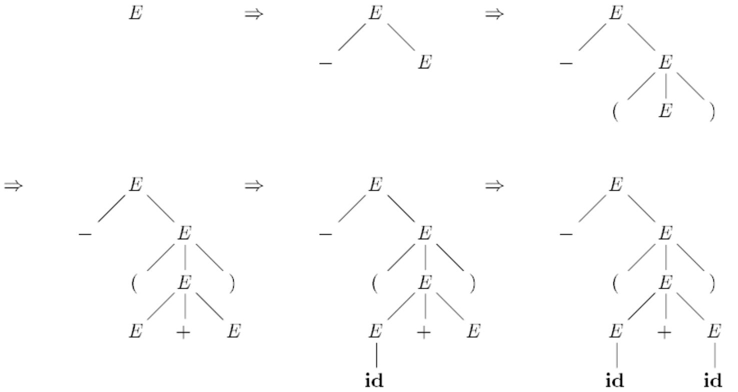


Figure 4.4: Sequence of parse trees for derivation (4.8)

Ambiguity

A grammar that produces more than one parse tree for some sentence is called ambiguous.

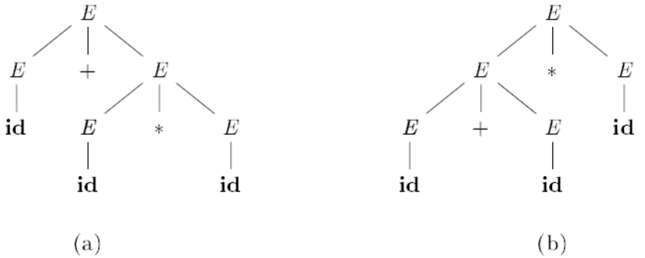


Figure 4.5: Two parse trees for **id+id∗id**

Problems: (1) Ambiguity can make parsing difficult; (2) Underlying structure is ill-defined.

Language Generated by a Grammar

A grammar G generates a language L if we can show that:

- Every string generated by G is in L, and
- Every string in L can be generated by G.

Example: Show that the grammar

$$S \to (S)S \mid \varepsilon$$

generates all strings of balanced parentheses and only such strings.

Grammars vs Regular Expressions

Every regular language is a context-free language but non vice-versa.

Example: The language generated by the regular expression

$$(a|b)^*abb$$

is equivalent to the grammar

$$\begin{array}{lcl} A_0 & \rightarrow & aA_0 \mid bA_0 \mid aA_1 \\ A_1 & \rightarrow & bA_2 \\ A_2 & \rightarrow & bA_3 \\ A_3 & \rightarrow & \varepsilon \end{array}$$

NFA-based Construction

From the NFA for the regular expression,

- For each state i of the NFA, create a nonterminal A_i
- Add production $A_i \to aA_j$ for each transition from i to j on a
- If i is accepting then add $A_i \to \varepsilon$
- If i is the starting state, make A_i the start symbol of the grammar.

Grammar with no Corresponding Regular Expression

The language

$$L = \{a^n b^n \mid n \geq 1\}$$

can be described by a grammar but not by a regular expression. Why?

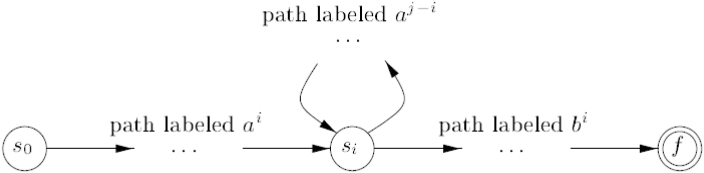


Figure 4.6: DFA D accepting both $a^i b^i$ and $a^j b^i$.

Non-Context-Free Grammars

Grammars alone can be not sufficient to specify some programming language construct.
This happens for constructs that are *context-dependent*.
The language

$$L_1 = \{wcw \mid w \text{ in } (\mathbf{a}|\mathbf{b})^*\}$$

is non-context-free. L_1 abstracts the requirements that identifiers are defined before their use (as in C and Java).

$$L_2 = \{a^n b^m c^n d^m \mid n \geq 0, m \geq 0\}$$

is non-context-free. L_2 abstracts the requirements that the number of formal parameters in a function declaration is the same as the number of actual parameters in a use of the function.

Common Grammars Problems (CGP)

A grammar may have some 'bad' styles or ambiguity. Some CGP are:

- Ambiguity
- Left-recursion
- Left factors

We need to transform a grammar G_1 into a grammar G_2 with no CGP and such that G_1 and G_2 are equivalent, i.e. they define the same language.

Eliminating Ambiguity

Consider the grammar:

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textbf{if } \text{expr } \textbf{then } \text{stmt} \\
& | & \textbf{if } \text{expr } \textbf{then } \text{stmt } \textbf{else } \text{stmt} \\
& | & \textbf{other}
\end{array}
$$

The sentence

$$\textbf{if } \text{E1 } \textbf{then } \textbf{if } \text{E2 } \textbf{then } \text{S1 } \textbf{else } \text{S2}$$

is ambiguous (cf. Figure 4.9).

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textit{matched\_stmt} \\
& | & \textit{open\_stmt} \\
\textit{matched\_stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{matched\_stmt} \textbf{ else } \textit{matched\_stmt} \\
& | & \textbf{other} \\
\textit{open\_stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
& | & \textbf{if } \textit{expr} \textbf{ then } \textit{matched\_stmt} \textbf{ else } \textit{open\_stmt}
\end{array}
$$

Figure 4.10: Unambiguous grammar for if-then-else statements

Example

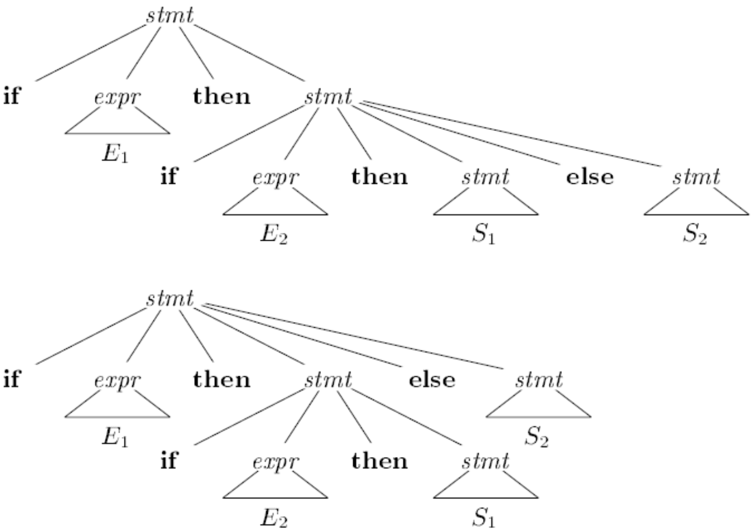


Figure 4.9: Two parse trees for an ambiguous sentence

CGP: Left Recursion

Definition

A grammar G is recursive if it contains a nonterminal X such that $X \Rightarrow^{+} \alpha X \beta$.
G is left-recursive if $X \Rightarrow^{+} X\beta$.
G is immediately left-recursive if $X \Rightarrow X\beta$.

Top-down parsing cannot handle left-recursive grammars.

We need to eliminate left recursion.

Eliminating Left Recursion

Consider a grammar G with a production

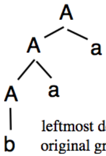
$$A \to A\alpha \mid \beta,$$

where β does not start with A.
Transform G in G' by replacing it by

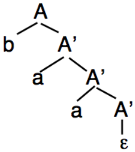
$$\begin{array}{rcl} A & \to & \beta A' \\ A' & \to & \alpha A' \mid \varepsilon. \end{array}$$

G and G' are equivalent: $L(G) = L(G')$.

input baa
$\beta \equiv b$
$\alpha \equiv a$



leftmost derivation
original grammar G



leftmost derivation
revised grammar G'

The Grammar Expression Example

The non-left-recursive expression grammar

$$
\begin{array}{rcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \mid \varepsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT' \mid \varepsilon \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

is obtained by eliminating immediate left recursion from the expression grammar

$$
\begin{array}{rcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

by applying the above transformation.

Algorithm for Eliminating Left Recursion

Input: A grammar G with no cycles and no ε-productions.
Output: An equivalent grammar with no left recursion..

arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
for (each i from 1 to n) {
    for (each j from 1 to $i-1$) {
        replace each production of the form $A_i \to A_j\gamma$ by the
          productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where
          $A_j \to \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current A_j-productions
    }
    eliminate the immediate left recursion among the A_i-productions
}

Figure 4.11: Algorithm to eliminate left recursion from a grammar

Applying the Algorithm

for $i = 1$ **to** n **do**

- **for** $j = 1$ **to** $i - 1$ **do**
  - ▷ *replace* $A_i \rightarrow A_j\gamma$
    with $A_i \rightarrow \delta_1\gamma \mid \cdots \mid \delta_k\gamma$
    where $A_j \rightarrow \delta_1 \mid \cdots \mid \delta_k$ *are all the current* A_j*-productions.*
- **Eliminate immediate left-recursion for** A_i
  - ▷ *New nonterminals generated above are numbered* A_{i+n}

- **Original Grammar:**
  - **(1)** $S \rightarrow Aa \mid b$
  - **(2)** $A \rightarrow Ac \mid Sd \mid e$
- **Ordering of nonterminals:** $S \equiv A_1$ **and** $A \equiv A_2$**.**
- $i = 1$
  - **do nothing as there is no immediate left-recursion for** S
- $i = 2$
  - **replace** $A \rightarrow Sd$ **by** $A \rightarrow Aad \mid bd$
  - **hence (2) becomes** $A \rightarrow Ac \mid Aad \mid bd \mid e$
  - **after removing immediate left-recursion:**
    - ▷ $A \rightarrow bdA' \mid eA'$
    - ▷ $A' \rightarrow cA' \mid adA' \mid \epsilon$
- **Resulting grammar:**
  - ▷ $S \rightarrow Aa \mid b$
  - ▷ $A \rightarrow bdA' \mid eA'$
  - ▷ $A' \rightarrow cA' \mid adA' \mid \epsilon$

CGP: Left Factor

The *left factor* problem occurs when for some nonterminal A there are A- productions whose bodies have a common prefix.
Example

$$\begin{array}{rcl} \mathit{stmt} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\ & | & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \end{array}$$

On input **if**, we have no way to decide which production to choose.

Idea: Expand with the full common factor!

Eliminating Left Factors

The algorithm below produces on input G an equivalent left-factored G'.

Input: context free grammar G

Output: equivalent left-factored context-free grammar G'

for each nonterminal A do

- **find the longest non-ϵ prefix α that is common to right-hand sides of two or more productions;**
- **replace**
  - ▷ $A \rightarrow \alpha\beta_1 \mid \cdots \mid \alpha\beta_n \mid \gamma_1 \mid \cdots \mid \gamma_m$

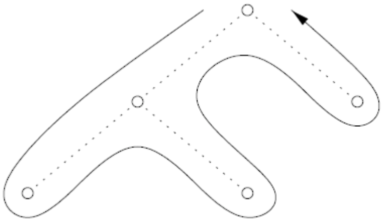
  with
  - ▷ $A \rightarrow \alpha A' \mid \gamma_1 \mid \cdots \mid \gamma_m$
  - ▷ $A' \rightarrow \beta_1 \mid \cdots \mid \beta_n$
- **repeat the above step until the grammar has no two productions with a common prefix;**

Top-down Parsing

Constructing a parse tree for the input string starting from the root in a depth-first manner (leftmost derivation).

```
procedure visit(node N) {
        for ( each child C of N, from left to right ) {
                visit(C);
        }
        evaluate semantic rules at node N;
}
```

Figure 2.11: A depth-first traversal of a tree



Example

Given the grammar

$$\begin{array}{rcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \varepsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \varepsilon \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

the sequence of trees given in the next slide corresponds to a leftmost derivation of the input string $\mathbf{id} + \mathbf{id} * \mathbf{id}$.

Example (ctdn.)

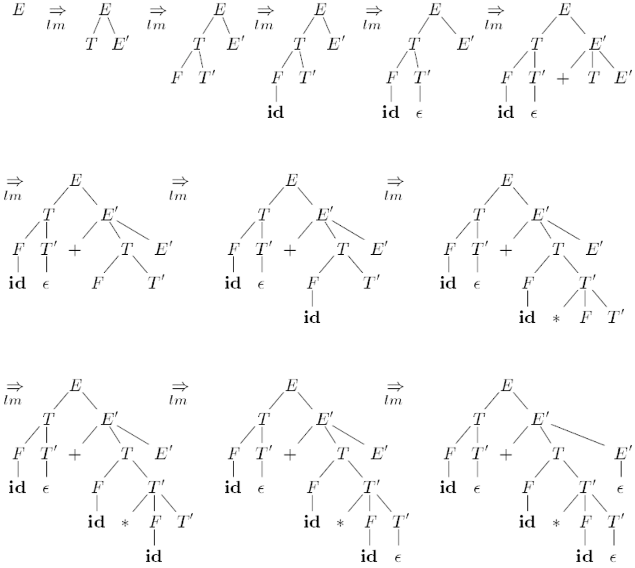


Figure 4.12: Top-down parse for $\mathbf{id} + \mathbf{id} * \mathbf{id}$

Recursive-descent Parsing

A recursive-descent parsing program is a set of procedures, one for each nonterminal, of the form:

```
void A() {
    Choose an A-production, A → X1 X2 ··· Xk;
    for ( i = 1 to k ) {
        if ( Xi is a nonterminal )
            call procedure Xi();
        else if ( Xi equals the current input symbol a )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}
```

Figure 4.13: A typical procedure for a nonterminal in a top-down parser

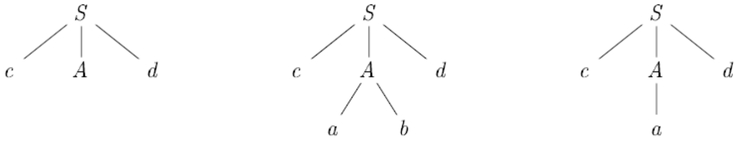
Backtracking

Top-down parsing may require repeated scans over the input: if an A-production leads to a failure, we must *backtrack* and try with another one.

Example

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

On input $w = cad$ we apply recursive-descent parsing. Since the choice of the first production leads to failure, we backtrack and try the second.



(a) (b) (c)

Predictive Parsing

The previous approach may be very inefficient due to backtracking.
A predictive parser is a recursive-descent parser needing no backtracking.
A predictive parser can choose one of the available productions for a nonterminal A by looking at the next input symbol(s).
The class of **LL(1)** grammars [Lewis&Stearns 1968] can parsed by a predictive parsers in $O(n)$ time.
We first need to introduce two important functions:
FIRST and FOLLOW.

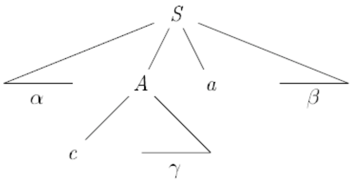


Figure 4.15: Terminal c is in FIRST(A) and a is in FOLLOW(A)

Computing FIRST

To compute FIRST(X) for any symbol X, apply the rules:

1. If X is a terminal, then FIRST(X) = {X}.
2. if X $\rightarrow$ ε is a production then place ε in FIRST(X)
3. If X is a nonterminal and X $\rightarrow$ Y_l Y_2 ... Y_k is a production for some k ≥ 1, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y_i), ... ,FIRST(Y_{i-1}); that is, Y_l ...Y_{i-1} $\Rightarrow^*$ ε. If ε is in FIRST(Y_j) for all j = 1,2, ... ,k, then add ε to FIRST(X).

Computing FIRST (ctd.)

Let $\alpha = X_1 X_2 \cdots X_n$. Perform the following steps in sequence:

- **FIRST$(\alpha)\Leftarrow$ FIRST$(X_1) - \{\epsilon\}$;**
- **if $\epsilon \in$ FIRST(X_1), then**
  - ▷ ***put FIRST$(X_2) - \{\epsilon\}$ into FIRST(α);***
- **if $\epsilon \in$ FIRST$(X_1) \cap$ FIRST(X_2), then**
  - ▷ ***put FIRST$(X_3) - \{\epsilon\}$ into FIRST(α);***
- $\cdots$
- **if $\epsilon \in \cap_{i=1}^{n-1}$FIRST$(X_i)$, then**
  - ▷ ***put FIRST$(X_n) - \{\epsilon\}$ into FIRST(α);***
- **if $\epsilon \in \cap_{i=1}^{n}$FIRST$(X_i)$, then**
  - ▷ ***put $\{\epsilon\}$ into FIRST(α).***

Computing FOLLOW

To compute Follow(X) for all nonterminal X, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), (S start symbol, \$ the input right endmarker).

2. If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε then everything in FOLLOW(A) is in FOLLOW(B).

3. If there is a production $A \rightarrow \alpha B\beta$ then everything in in FIRST(β) except ε is in FOLLOW(B).

FIRST and FOLLOW Example

$E \to T\ E'$
$E' \to +\ T\ E' \mid \varepsilon$
$T \to F\ T'$
$T' \to *\ F\ T' \mid \varepsilon$
$F \to (\ E\ ) \mid \textbf{id}$

1. If X is a terminal, then FIRST(X) = {X}.

2. If X is a nonterminal and $X \Rightarrow Y_l\ Y_2 \ldots Y_k$ is a production for some k > 1, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y_l), ... ,FIRST(Y_{i-1}); that is, $Y_l \ldots Y_{i-1} \Rightarrow \varepsilon$. If ε is in FIRST($Y_j$) for all j = 1,2, ... ,k, then add ε to FIRST(X).

Computing FOLLOW(A)

- Place \$ into FOLLOW(S)
- Repeat until nothing changes:
  - if $A \to \alpha B\beta$ then add FIRST(β)\{ε} to FOLLOW(B)
  - if $A \to \alpha B$ then add FOLLOW(A) to FOLLOW(B)
  - if $A \to \alpha B\beta$ and ε is in FIRST(β) then add FOLLOW(A) to FOLLOW(B)

- FIRST(F) = FIRST(T) = FIRST(E) = {(, id }
- FIRST(E') = {+, ε}
- FIRST(T') = {*, ε}
- FOLLOW(E) = FOLLOW(E') = {), \$}
- FOLLOW(T) = FOLLOW(T') = {+,),\$}
- FOLLOW(F) = {+, *,), \$}

LL(1) Grammars

Left to right parsers producing a **L**eftmost derivation *looking* ahead by at most **1** input symbol.

Definition

A grammar G is **LL(1)** if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions in G, then

- FIRST(α) and FIRST(β) are disjoint sets
- If ε is in FIRST(β) then FIRST(α) and FOLLOW(A) are disjoint sets
- If ε is in FIRST(α) then FIRST(β) and FOLLOW(A) are disjoint sets.

Most programming language constructs are **LL(1)** but careful grammar writing is required.
If a grammar is **LL(1)** then it does not have CGP, but the vice-versa does not hold.

Predictive Parsing Table

To construct a parsing table M for a grammar G, for each production $A \rightarrow \alpha$ in G:

- If a is in FIRST(a), add $A \rightarrow \alpha$ in $M[A, a]$.
- If ε is in FIRST(α), add $A \rightarrow \alpha$ in $M[A, b]$ for each b in FOLLOW(A).
- If ε is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ in $M[A, \$]$.

Example

For the expression grammar the algorithm produces the following table.

NON - TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Figure 4.17: Parsing table M for Example 4.32

Stack-based Predictive Parser

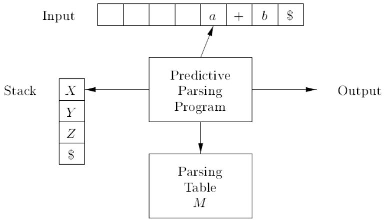


Figure 4.19: Model of a table-driven predictive parser

let a be the first symbol of w;
let X be the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
    if ($X = a$) pop the stack and **let** a be the next symbol of w;
    else if (X is a terminal) *error*();
    else if ($M[X, a]$ is an error entry) *error*();
    else if ($M[X, a] = X \to Y_1 Y_2 \cdots Y_k$) {
        output the production $X \to Y_1 Y_2 \cdots Y_k$;
        pop the stack;
        push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
    }
    let X be the top stack symbol;
}

Figure 4.20: Predictive parsing algorithm

Example

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to +\ TE'$
$\mathbf{id}\ +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id}\ +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id}\ +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$*\ FT'E'\$$	$* \mathbf{id}\$$	output $T' \to *\ FT'$
$\mathbf{id} + \mathbf{id}\ *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id}\ *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Figure 4.21: Moves made by a predictive parser on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$