Front End: Syntax Analysis

The Role of the Parser

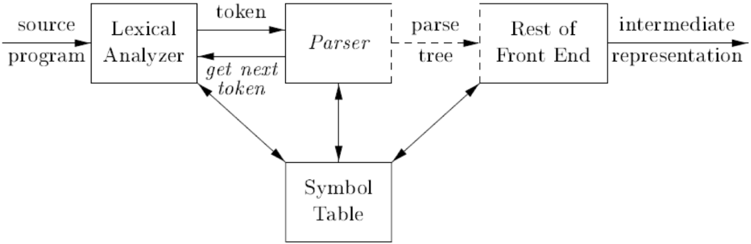


Figure 4.1: Position of parser in compiler model

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.

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

  (associativity and precedence captured)
- Top-down methods: **LL** grammars, e.g.

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

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

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

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

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

Rightmost derivation: always choose the righttmost nonterminal

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

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.

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

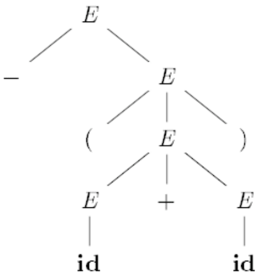


Figure 4.3: Parse tree for $-(\textbf{id} + \textbf{id})$

Example

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

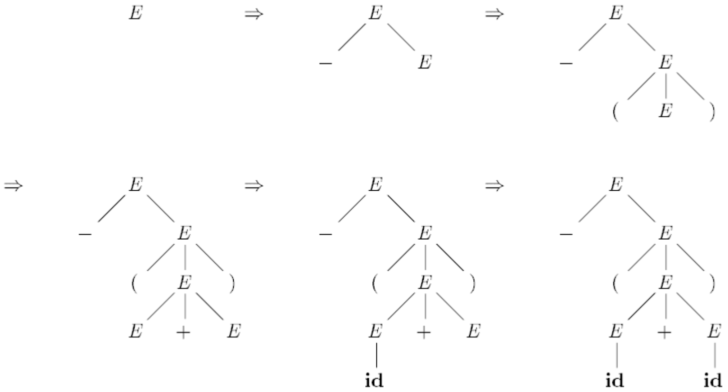


Figure 4.4: Sequence of parse trees for derivation (4.8)

Ambiguity

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

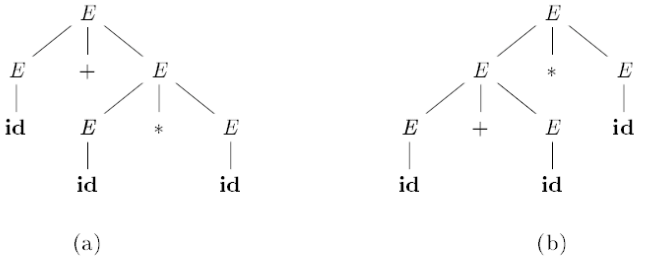


Figure 4.5: Two parse trees for **id+id∗id**

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 $A_i \rightarrow aA_j$ for each transition from i to j on a
- If i is accepting then add $A_i \rightarrow \varepsilon$
- 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?

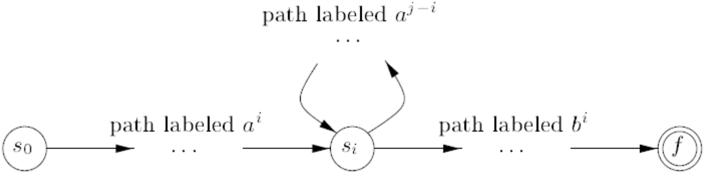


Figure 4.6: DFA D accepting both $a^i b^i$ and $a^j b^i$.

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

if E1 **then** **if** E2 **then** S1 **else** 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

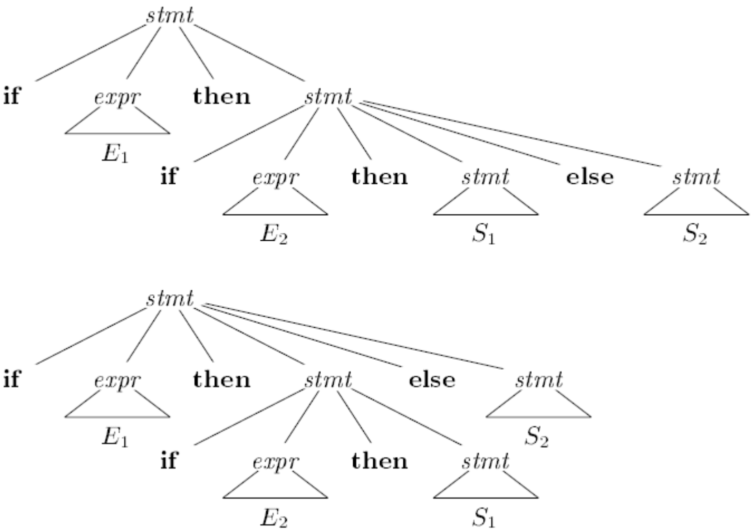


Figure 4.9: Two parse trees for an ambiguous sentence

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

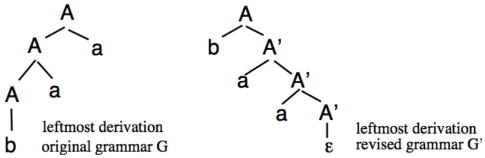
$$A \to A\alpha \mid \beta,$$

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

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

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

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



The Grammar Expression Example

The non-left-recursive expression grammar

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

is obtained by eliminating immediate left recursion from the expression grammar

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

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..

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3)   **for** (each j from 1 to $i - 1$) {
4)   replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current A_j-productions
5)   }
6)   eliminate the immediate left recursion among the A_i-productions
7) }

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*** $A_i \to A_j\gamma$
    with $A_i \to \delta_1\gamma \mid \cdots \mid \delta_k\gamma$
    where $A_j \to \delta_1 \mid \cdots \mid \delta_k$ ***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)** $S \to Aa \mid b$
  - **(2)** $A \to Ac \mid Sd \mid e$
- **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** $A \to Sd$ **by** $A \to Aad \mid bd$
  - **hence (2) becomes** $A \to Ac \mid Aad \mid bd \mid e$
  - **after removing immediate left-recursion:**
    - ▷ $A \to bdA' \mid eA'$
    - ▷ $A' \to cA' \mid adA' \mid \epsilon$
- **Resulting grammar:**
  - ▷ $S \to Aa \mid b$
  - ▷ $A \to bdA' \mid eA'$
  - ▷ $A' \to cA' \mid adA' \mid \epsilon$

CGP: Left Factor

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

$$\begin{array}{lcl} \textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\ & \mid & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \end{array}$$

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**
  - ▷ $A \rightarrow \alpha\beta_1 \mid \cdots \mid \alpha\beta_n \mid \gamma_1 \mid \cdots \mid \gamma_m$

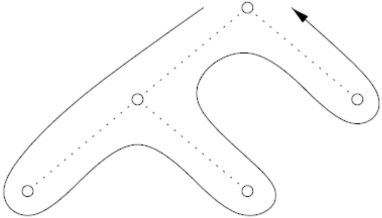
  with
  - ▷ $A \rightarrow \alpha A' \mid \gamma_1 \mid \cdots \mid \gamma_m$
  - ▷ $A' \rightarrow \beta_1 \mid \cdots \mid \beta_n$
- **repeat the above step until the grammar has no two productions with a common prefix;**

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.)

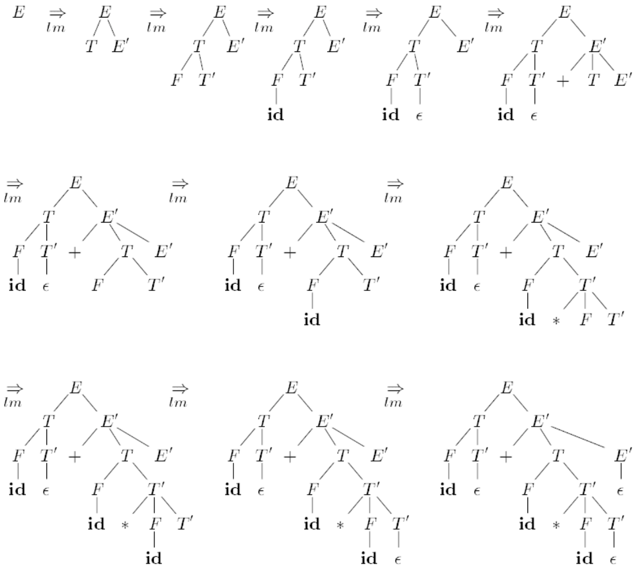


Figure 4.12: Top-down parse for **id** + **id** ∗ **id**

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 → X_1 X_2 ··· X_k;
    for ( i = 1 to k ) {
        if ( X_i is a nonterminal )
            call procedure X_i();
        else if ( X_i 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

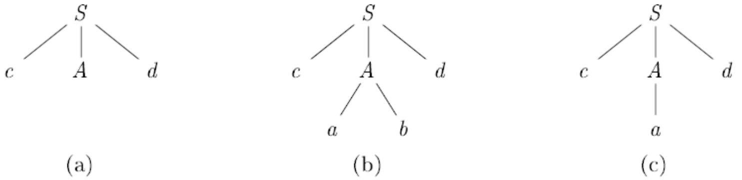
Backtracking

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.



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 be parsed by a predictive parsers in $O(n)$ time.
We first need to introduce two important functions:
FIRST and FOLLOW.

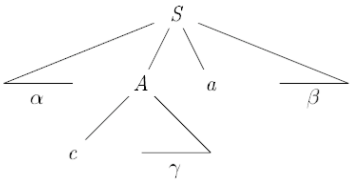


Figure 4.15: Terminal c is in FIRST(A) and a is in FOLLOW(A)

FIRST

Definition

Let G be a grammar and let α be a string on $T \cup N$.

$\textsc{First}(\alpha)$ is the set of terminal symbols that may occur at the beginning of a string derived from α:

$a \in T$, $a \in \textsc{First}(\alpha)$ if and only if $\alpha \Rightarrow^* a\beta$ for some $\beta \in (T \cup N)^*$.

If $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in \textsc{First}(\alpha)$.

FOLLOW

Definition

Let G be a grammar and let A be a non-terminal of G.

$\textsc{Follow}(A)$ is the set of terminal symbols that may occur on the right hand side immediately after A in a sentential form:

$a \in T$, $a \in \textsc{Follow}(A)$ if and only if $S \Rightarrow^* \alpha A a \beta$ for some $\alpha, \beta \in (T \cup N)^*$.

If $S \Rightarrow^* \alpha A$, then $\$ \in \textsc{Follow}(A)$.

Computing FIRST

To compute $\textsc{First}(()X)$ for any symbol X, apply the rules:

1. If X is a terminal, then FIRST(X) = {X}.
2. if X → ε is a production then place ε in FIRST(X)
3. If X is a nonterminal and X → Y_1 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_1 ...Y_{i-1} ⇒* ε. If ε is in FIRST(Y_j) for all j = 1,2, ... ,k, then add ε to FIRST(X).

Computing FIRST (ctd.)

To compute $\text{FIRST}(\alpha)$ for any string of symbol α, apply the rules:

Let $\alpha = X_1X_2\cdots X_n$. 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 FIRST: Example

Example for computing FIRST(α)

Grammar

$E \to E'T$

$E' \to -TE' \mid \epsilon$

$T \to FT'$

$T' \to /FT' \mid \epsilon$

$F \to int \mid (E)$

$\textbf{\textit{FIRST}}(F) = \{int, (\}$

$\textbf{\textit{FIRST}}(T') = \{/, \epsilon\}$

$\textbf{\textit{FIRST}}(T) = \{int, (\}$

$\textbf{\textit{FIRST}}(E') = \{-, \epsilon\}$

$\textbf{\textit{FIRST}}(E) = \{-, int, (\}$

$\textbf{\textit{FIRST}}(E'T) = \{-, int, (\}$

$\textbf{\textit{FIRST}}(-TE') = \{-\}$

$\textbf{\textit{FIRST}}(\epsilon) = \{\epsilon\}$

$\textbf{\textit{FIRST}}(FT') = \{int, (\}$

$\textbf{\textit{FIRST}}(/FT') = \{/\}$

$\textbf{\textit{FIRST}}(\epsilon) = \{\epsilon\}$

$\textbf{\textit{FIRST}}(int) = \{int\}$

$\textbf{\textit{FIRST}}((E)) = \{(\}$

- **FIRST**$(T'E') =$
  - $(\textbf{\textit{FIRST}}(T') - \{\epsilon\})\cup$
  - $(\textbf{\textit{FIRST}}(E') - \{\epsilon\})\cup$
  - $\{\epsilon\}$

Computing FOLLOW

To compute FOLLOW(X) for all nonterminals 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 → T E'
E'→ + T E' | ε
T → F T'
T'→ * F T' | ε
F → (E) | **id**

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

2. If X is a nonterminal and X ⇒ 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_l), ... ,FIRST(Y_{i-1}); that is, Y_l ...Y_{i-1} ⇒ ε. 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 → αBβ then add FIRST(β)\{ε} to FOLLOW(B)
  - if A → αB then add FOLLOW(A) to FOLLOW(B)
  - if A → αBβ 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) = {+, *,), $}

Another FIRST and FOLLOW Example

Consider the grammar:

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & \epsilon \mid +E \mid -E \\
T & \rightarrow & AT' \\
T' & \rightarrow & \epsilon \mid *T \\
A & \rightarrow & \mathbf{a} \mid \mathbf{b} \mid (E)
\end{array}
$$

Computing $\textsc{First}(X)$ and $\textsc{Follow}(X)$ for all X in the grammar gives the following result:

	$\textsc{First}()$	$\textsc{Follow}()$
E	$\mathbf{a}, \mathbf{b}, ($	$\$,)$
E'	$\epsilon, +, -$	$\$,)$
T	$\mathbf{a}, \mathbf{b}, ($	$\$,), +, -$
T'	$\epsilon, *$	$\$,), +, -$
A	$\mathbf{a}, \mathbf{b}, ($	$\$,), +, -, *$

How Predictive Parsers Work

Consider a predictive parser implemented as a *non-recursive* procedure that explicitly operates on a stack.
INIT: parser pushes the start symbol on the stack and call the scanner to get the first token.
LOOP:

- if TOP is $X \in N$, then
  - Choose a production $X \to \beta$ (looking at the current token)
  - Pop X and push β (from right to left).
  - Goto LOOP.
- If TOP is $a \in T$ and a matches the current token
  - Pop a and ask scanner for the next token
  - Goto LOOP.
- If STACK is empty and there are no more tokens, **ACCEPT**!
- If none of the above hold, **FAIL**!

Why computing FIRST?

Suppose that during parsing

- TOP is a non-terminal X and

$$X \rightarrow \alpha_1, \ldots, X \rightarrow \alpha_k$$

  are all productions in the string grammar.
- The current lookahead token is a
- $a \in \textsc{First}(\alpha_i)$ for more than one i.

Then the parser cannot choose deterministically and may need to backtrack.

Why computing FOLLOW?

Suppose that during parsing

- TOP is a non-terminal X and

$$X \to \alpha_1, \ldots, X \to \alpha_k$$

  are all productions in the string grammar.
- The current lookahead token is a.
- $a \notin \textsc{First}(\alpha_i)$ for all i's.

Then the parser can still select a production to expand X:
If $\alpha_i \Rightarrow^* \varepsilon$, for some i, and $a \in \textsc{Follow}(X)$, the production $X \to \alpha_i$ is a suitable one.
Note that $\alpha_i \Rightarrow^* \varepsilon$ iff $\varepsilon \in \textsc{First}(\alpha_i)$.

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 $A \rightarrow \alpha \mid \beta$ 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.

(Non) Example

Is the following grammar **LL(1)**?

$$
\begin{aligned}
G &\rightarrow aAb \mid aBbb \\
A &\rightarrow aAb \mid 0 \\
B &\rightarrow aBbb \mid 1
\end{aligned}
$$

No: it is not factored.

$$
\begin{aligned}
G &\rightarrow aG' \\
G' &\rightarrow Ab \mid Bbb \\
A &\rightarrow aAb \mid 0 \\
B &\rightarrow aBbb \mid 1
\end{aligned}
$$

This factored version is still not **LL(1)**. Why?

LL (Predictive) Parsing Table

A Predictive Parsing Table is a bidimensional matrix M where

- Rows represent non-terminals
- Columns represent terminals (including \$), and
- $M[A, a]$ contains the productions chosen for expanding A with a as the current input.

Predictive Parsing Table

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

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

An empty entry in M corresponds to an error.

Definition

A grammar is **LL(1)** if and only if every entry of the parsing table contains *at most* una production.

Example I

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

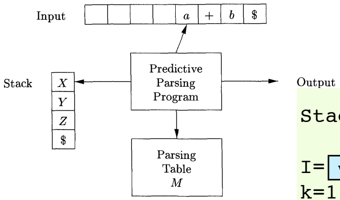
Example II

$$
\begin{aligned}
S &\rightarrow iEtSS' \mid a \\
S' &\rightarrow eS \mid \varepsilon \\
E &\rightarrow b
\end{aligned}
$$

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$\$$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Figure 4.18: Parsing table M for Example 4.33

Table-driven Predictive Parser




```
Stack= [S][$];
I=[w][$];
k=1;
X = top();
while(X <> $){ //stack non empty
  if (X == I[k]) {pop(); k++;}
  else if (X is a terminal)
                   error();
  else if (M[X,I[k]] == error)
                   error();
  else if (M[X,I[k]] == X→Y1...Yn){
    output_production(X→Y1...Yn);
    pop();
    push(Yn);...;push(Y1);
  }
  X=top();
}
```

Example

Matched	Stack	Input	Action
	$E\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	
	$TE'\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	output $T \to FT'$
	$\textbf{id}\ T'E'\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	output $F \to \textbf{id}$
$\textbf{id}$	$T'E'\$$	$+ \textbf{id} * \textbf{id}\$$	match $\textbf{id}$
$\textbf{id}$	$E'\$$	$+ \textbf{id} * \textbf{id}\$$	output $T' \to \epsilon$
$\textbf{id}$	$+\ TE'\$$	$+ \textbf{id} * \textbf{id}\$$	output $E' \to +\ TE'$
$\textbf{id}\ +$	$TE'\$$	$\textbf{id} * \textbf{id}\$$	match $+$
$\textbf{id}\ +$	$FT'E'\$$	$\textbf{id} * \textbf{id}\$$	output $T \to FT'$
$\textbf{id}\ +$	$\textbf{id}\ T'E'\$$	$\textbf{id} * \textbf{id}\$$	output $F \to \textbf{id}$
$\textbf{id} + \textbf{id}$	$T'E'\$$	$* \textbf{id}\$$	match $\textbf{id}$
$\textbf{id} + \textbf{id}$	$*\ FT'E'\$$	$* \textbf{id}\$$	output $T' \to *\ FT'$
$\textbf{id} + \textbf{id}\ *$	$FT'E'\$$	$\textbf{id}\$$	match $*$
$\textbf{id} + \textbf{id}\ *$	$\textbf{id}\ T'E'\$$	$\textbf{id}\$$	output $F \to \textbf{id}$
$\textbf{id} + \textbf{id} * \textbf{id}$	$T'E'\$$	$\$$	match $\textbf{id}$
$\textbf{id} + \textbf{id} * \textbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\textbf{id} + \textbf{id} * \textbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

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

More Examples

$$
\begin{aligned}
S &\rightarrow aAB \\
A &\rightarrow C \mid D \\
B &\rightarrow b \\
C &\rightarrow c \mid \epsilon \\
D &\rightarrow d
\end{aligned}
$$

	FIRST()	FOLLOW()
S	a	$
A	c, d, ϵ	b
B	b	$
C	c, ϵ	b
D	d	b

	a	b	c	d	$
S	$S \rightarrow aAB$				
A		$A \rightarrow C$	$A \rightarrow C$	$A \rightarrow D$	
B		$B \rightarrow b$			
C		$C \rightarrow \epsilon$	$C \rightarrow c$		
D				$D \rightarrow d$	

OUTPUT	PILA	INPUT
Start	$S\$$	$adb\$$
$S \rightarrow aAB$	$aAB\$$	$adb\$$
	$AB\$$	$db\$$
$A \rightarrow D$	$DB\$$	$db\$$
$D \rightarrow d$	$dB\$$	$db\$$
	$B\$$	$b\$$
$B \rightarrow b$	$b\$$	$b\$$
	$	$
OK!		

OUTPUT	PILA	INPUT
Start	$S\$$	$abb\$$
$S \rightarrow aAB$	$aAB\$$	$abb\$$
	$AB\$$	$bb\$$
$A \rightarrow C$	$CB\$$	$bb\$$
$C \rightarrow \epsilon$	$B\$$	$bb\$$
$B \rightarrow b$	$b\$$	$bb\$$
	$	$b\$$
Errore!		