Syntax Analysis, Parsing

Lex – example-1
Input file – **input_first**

if + 78 else 0

Tokens: if, else, op (+,-), number, other

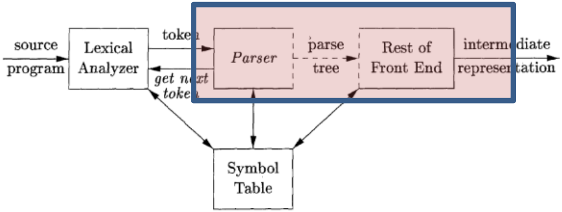
Parsing

- Every programming language **has precise grammar rules** that describe the **syntactic structure** of well-formed programs
  - In C, the **rules** states a **program consists of functions**, a **function consist of declarations and statements**, a **statement consists of expressions**, and so on.

- The **task of a parser** is to

(a) **Obtains strings of tokens** from the lexical analyzer and **verifies** that the string follows **the rules of the source language**

(b) Parser **reports errors** and sometimes **recovers** from it



- **Type checking, semantic analysis** and **translation** actions can be **interlinked** with parsing
- Implemented as a **single module**.

Parsing

- Two major classes of parsing
    - top-down and bottom-up
- **Input** to the parser is **scanned from left to right**, one symbol at a time.

$$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle$$

- The **syntax of programming language** constructs can be specified by **context-free grammars**
- Grammars systematically describe the **syntax** of programming language constructs like **expressions and statements**.

$$stmt \ \rightarrow \ \mathbf{if} \ (\ expr \ ) \ stmt \ \mathbf{else} \ stmt$$

    - Quick recall

Context free grammar

- A CFG is denoted as $G = (N, T, P, S)$

  N : Finite set of non-terminals -- **syntactic variables** (stmt, expr)

  T : Finite set of terminals ---- **Tokens,** basic symbols from which strings and programs are formed

  S : The start symbol -- set of strings it generates is the **language** generated by the grammar

  P : Finite set of productions -- specify the manner in which the **terminals and nonterminals can be combined** to form strings

In this grammar, the terminal symbols are

id + - * / ( )

Productions

Start symbol:

expression

head → body

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

Task of a parser

Output of the parser is some **representation of the parse tree** for the **stream of tokens as input,** that comes from the lexical analyzer**.**

- **Top-down** parser works for **LL grammar**
- **Bottom-up** parser works for **LR grammars**
- Only subclasses of grammars
  - But expressive enough to describe **most of the syntactic constructs** of modern programming languages.

Concentrate on parsing expressions

- Constructs that begin with keywords like **while** or **int** are relatively easy to parse
  - because the **keyword guides** the parsing decisions
- We therefore **concentrate on expressions**, which present **more of challenge**, because of the **associativity and precedence** of operators

Derivations

The **construction of a parse tree** can be conceptualized as **derivations**

Derivation: Beginning with the **start symbol**, each rewriting step **replaces a nonterminal** by the body of one of its **productions**.

$$\alpha A \beta \Rightarrow \alpha \gamma \beta \qquad A \rightarrow \gamma \text{ is a production}$$

If $S \overset{*}{\Rightarrow} \alpha$, where S is the start symbol of a grammar G, we say that α is a *sentential form* of G.

A **sentence** of G is a sentential form with **no nonterminals**.
The language L(G) generated by a grammar G is its **set of sentences**.

Derivations

The **construction of a parse tree** can be conceptualized as **derivations**

Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions. $\alpha A\beta \Rightarrow \alpha\gamma\beta.\quad A \rightarrow \gamma$ is a production

Consider a grammar G

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

Derivation

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

1. Derivation of **–(id+id) from start symbol E**
2. –(id+id) is a **sentence** of G
3. At **each step** in a derivation, there are **two choices** to be made.
   - **Which nonterminal** to replace? **: leftmost derivations**
   - Accordingly we must **choose** a production

Derivations-- Rightmost derivations

Consider a grammar G

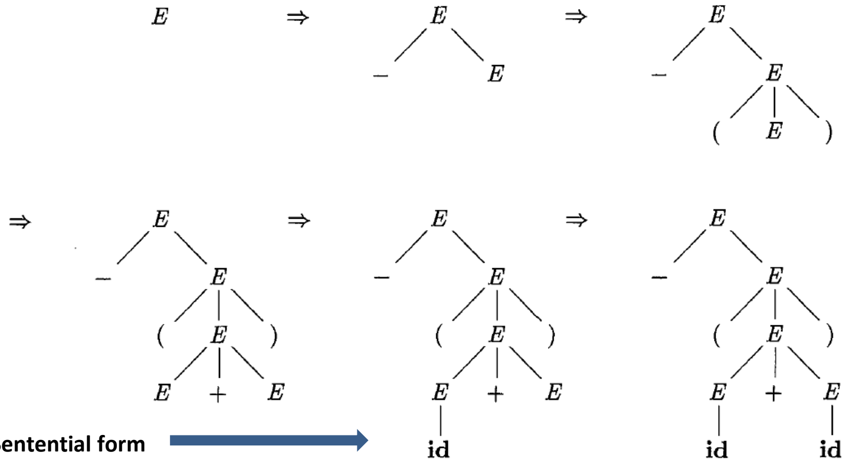
$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$$

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

1. Derivation of –(id+id) from E
2. –(id+id) is a sentence of G
3. At each step in a derivation, there are two choices to be made.
    - Which nonterminal to replace?
    - Accordingly we must pick a production → **Rightmost derivations**,

Parse trees

- A parse tree is a **graphical representation of a derivation** that exhibits
  - the **order** in which **productions are applied** to replace non-terminals
- The **internal node** is a **non-terminal** ***A*** in the head of the production
  - The **children of the node** are labelled, from left to right, by the symbols in the **body of the production** by which *A* was replaced during the derivation
- **Same parse tree** for leftmost and rightmost derivations



Sentential form (leaves of a parse tree)

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

parse tree for **- (id + id)**

Ambiguity

- A grammar that **produces more than one parse tree** for some **sentence** is said to be ***ambiguous***
- *An ambiguous grammar is one that produces **more than one leftmost derivation** or **more than one rightmost derivation** for the **same sentence**.*

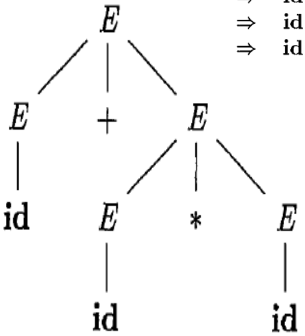
$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

$$\begin{array}{lll} E & \Rightarrow & E + E \\ & \Rightarrow & \mathbf{id} + E \\ & \Rightarrow & \mathbf{id} + E * E \\ & \Rightarrow & \mathbf{id} + \mathbf{id} * E \\ & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id} \end{array} \qquad \begin{array}{lll} E & \Rightarrow & E * E \\ & \Rightarrow & E + E * E \\ & \Rightarrow & \mathbf{id} + E * E \\ & \Rightarrow & \mathbf{id} + \mathbf{id} * E \\ & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id} \end{array}$$

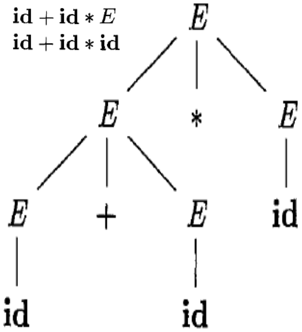
Two distinct leftmost derivations for the sentence **id + id * id**

Ambiguity

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow \mathbf{id} + E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$



(a)



(b)

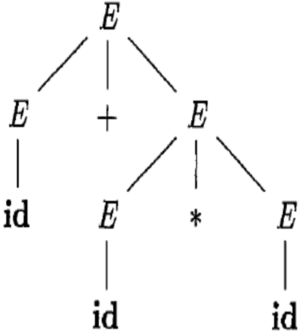
Two parse trees for **id+id*id**

Ambiguity

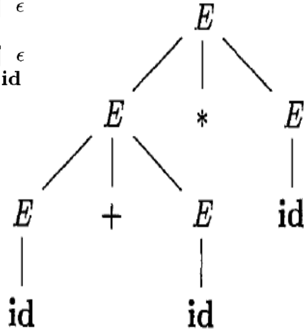
$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

Unambiguous grammar

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$



(a)



(b)

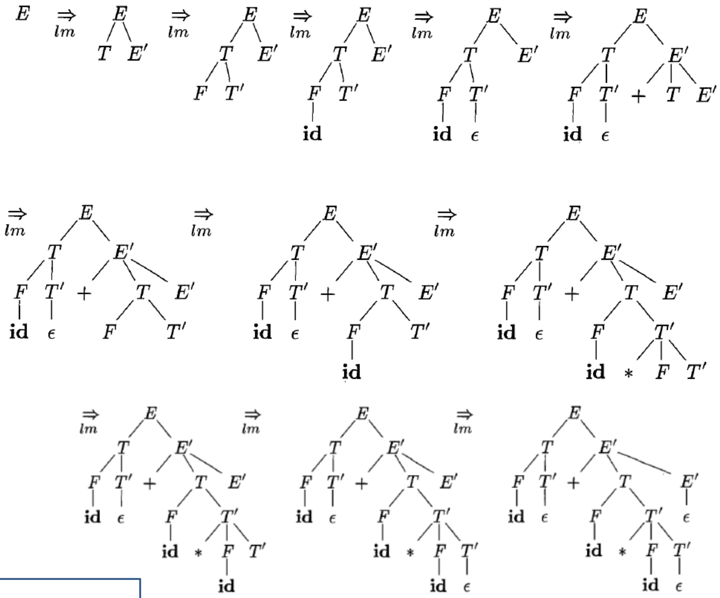
Two parse trees for **id+id*id**

Top-Down Parsing

- Top-down parsing can be viewed as the problem of
- **Constructing a parse tree** for the input string,
    - **starting from the root** and creating the nodes of the parse tree in **preorder**
- Top-down parsing can be viewed as finding a **leftmost derivation** for an input string

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \ \mid\ \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \ \mid\ \epsilon \\
F & \rightarrow & (\ E\ ) \ \mid\ \mathbf{id}
\end{array}
$$

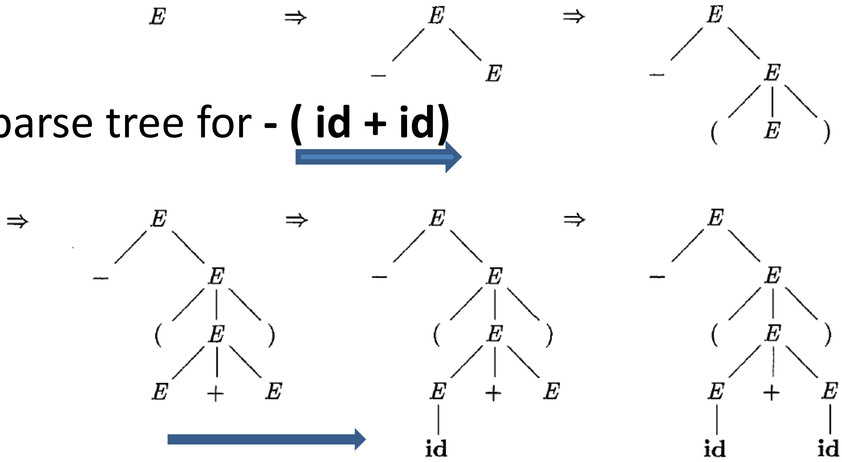
$\Longrightarrow$ **id+id*id**



id+id*id

Top-Down Parsing

parse tree for **- (id + id)**



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

parse tree for **- (+ id) ???**

Top-Down Parsing

A grammar is *left recursive* if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α. Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left

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

Left recursive

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

Non-Left recursive

Top-Down Parsing

Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

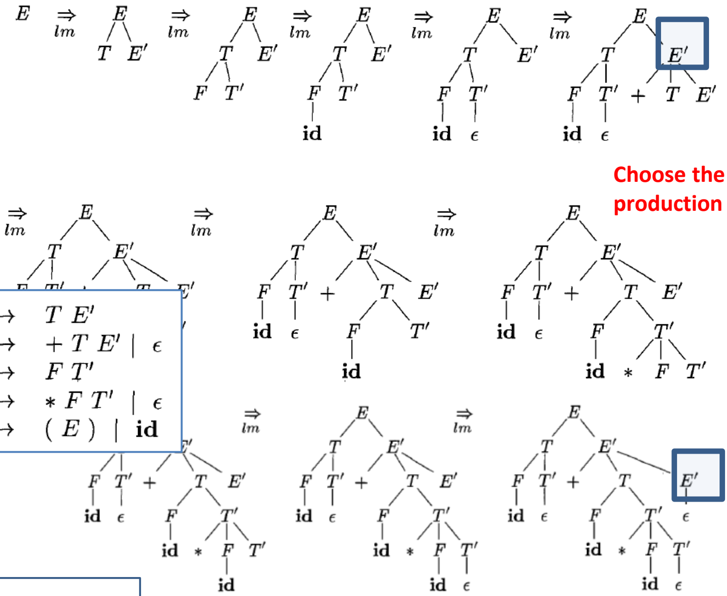
OUTPUT: An equivalent grammar with no left recursion.

```
1)  arrange the nonterminals in some order A_1, A_2, ..., 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 → A_j γ by the
                    productions A_i → δ_1 γ | δ_2 γ | ··· | δ_k γ, where
                    A_j → δ_1 | δ_2 | ··· | δ_k are all current A_j-productions
5)        }
6)        eliminate the immediate left recursion among the A_i-productions
7)  }
```

Top-Down Parsing

Challenges:
At **each step** of a top-down parse, the key problem is that of **determining the production to be applied** for a nonterminal, say A.

(a) **Recursive descent parsing**: May require **backtracking** to find the **correct A-production** to be applied
(b) **Predictive parsing:** No backtracking!
looking ahead at the input a fixed number of symbols (next symbols) – LL(k), LL(1) grammars



Choose the correct production

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

$$\boxed{\mathbf{id}+\mathbf{id}*\mathbf{id}}$$

Recursive-Descent Parsing

Nondeterministic

```
void A() {
    Choose an A-production, A → X₁X₂···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 */;
    }
}
```

Try other productions!

(a) A recursive-descent parsing **consists of a set of procedures**, one for each **nonterminal**.

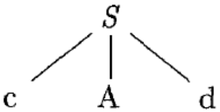
(b) Execution begins with the **procedure for the start symbol S**,

(c) Halts and announces success if **S() returns** and its procedure body scans the **entire input string**.

(d) Backtracking: may require **repeated scans over the input**

$$
\begin{aligned}
S &\rightarrow c\ A\ d \\
A &\rightarrow a\ b \mid a
\end{aligned}
$$

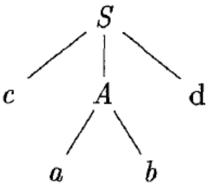
$$\boxed{\text{input string } w = cad,}$$



The leftmost leaf, **labeled c**, matches the first symbol of input **w (i.e. c)**, so we advance the input pointer to **a**

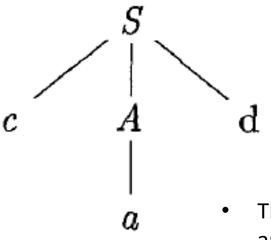
$$\boxed{\text{input string } w = cad,}$$

Now, we expand A using the first alternative $A \rightarrow a\ b$



- We have a match for the **second input symbol, a**,
- So we advance the **input pointer to d**, the third input symbol
- Compare ***d*** against the next leaf, labeled ***b***

Failure !! Backtrack!



(c)

input string $w = cad$,

we must reset the input pointer to position **a**

- The leaf **a** matches the second input symbol of w (i.e. **a**) and the leaf **d** matches the third input symbol **d**
- Since **S() returns** and we have **scanned w and produced a parse tree for w**,
- We halt and **announce successful completion of parsing**

Left Factoring

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

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

$$\begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \end{array}$$

Left factoring a grammar.

Top-Down Parsing

Challenges:
At **each step** of a top-down parse, the key problem is that of **determining the production to be applied** for a nonterminal, say A.

(a) Recursive descent parsing: May require backtracking to find the correct A-production to be applied
(b) **Predictive parsing:** No backtracking!
looking ahead at the input a fixed number of symbols (**next symbols**) –
LL(k), LL(1) grammars

Basic concept of Predictive parsing

Input string w=abcd

One sentential form

S=> aXY....

Grammar productions

1. X-> **b**A...

First symbol

2. X->cP ......

Another sentential form

S=> aXb

Grammar productions

1. X-> €
2. X-> ......

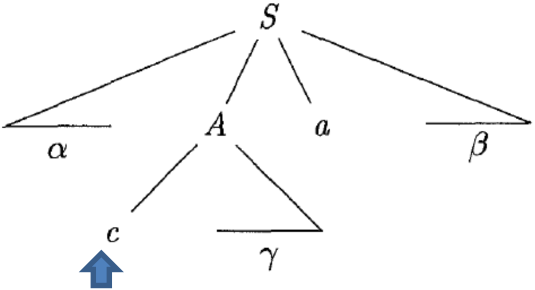
We know that **b** **Follows** **X** in any sentential form

4.4.2 FIRST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G. During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol. During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

Define $FIRST(\alpha)$, where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α. If $\alpha \overset{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST(α). For example, in Fig. 4.15, $A \overset{*}{\Rightarrow} c\gamma$, so c is in FIRST(A).

For a preview of how FIRST can be used during predictive parsing, consider two A-productions $A \to \alpha \mid \beta$, where FIRST(α) and FIRST(β) are disjoint sets. We can then choose between these A-productions by looking at the next input symbol a, since a can be in at most one of FIRST(α) and FIRST(β), not both. For instance, if a is in FIRST(β) choose the production $A \to \beta$. This idea will



First(A)

How to compute First(X)

To compute $\text{FIRST}(X)$ for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

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

2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \ldots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ, then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.

3. If $X \to \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

1. $\text{FIRST}(F) = \text{FIRST}(T) = \text{FIRST}(E) = \{(, \mathbf{id}\}$. To see why, note that the two productions for F have bodies that start with these two terminal symbols, **id** and the left parenthesis. T has only one production, and its body starts with F. Since F does not derive ϵ, $\text{FIRST}(T)$ must be the same as $\text{FIRST}(F)$. The same argument covers $\text{FIRST}(E)$.

2. $\text{FIRST}(E') = \{+, \epsilon\}$. The reason is that one of the two productions for E' has a body that begins with terminal $+$, and the other's body is ϵ. Whenever a nonterminal derives ϵ, we place ϵ in FIRST for that nonterminal.

3. $\text{FIRST}(T') = \{*, \epsilon\}$. The reasoning is analogous to that for $\text{FIRST}(E')$.

Basic concept of Predictive parsing

Input string w=abcd

One sentential form

S=> aXY….

Grammar productions

1. X-> **b**A…

First symbol

2. X->cP ……

Another sentential form

S=> aXb

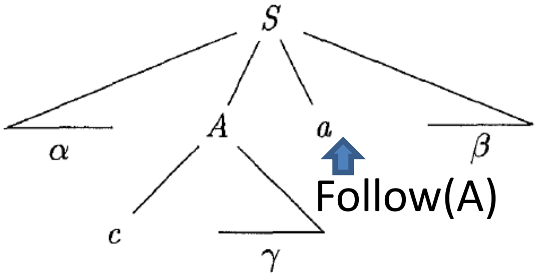
Grammar productions

1. X-> €
2. X-> ……

We know that **b** **Follows** **X** in any sentential form

FIRST and FOLLOW

Define $FOLLOW(A)$, for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form $S \stackrel{*}{\Rightarrow} \alpha A a \beta$, for some α and β, as in Fig. 4.15. Note that there may have been symbols between A and a, at some time during the derivation, but if so, they derived ϵ and disappeared. In addition, if A can be the rightmost symbol in some sentential form, then \$ is in FOLLOW(A); recall that \$ is a special "endmarker" symbol that is assumed not to be a symbol of any grammar.


S
A
a
α
β
c
γ
Follow(A)

How to compute Follow(A)

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

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

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

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

S-> xAyz   y in Follow(A)

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

$\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{), \$\}$. Since E is the start symbol, $\text{FOLLOW}(E)$ must contain \$. The production body (E) explains why the right parenthesis is in $\text{FOLLOW}(E)$. For E', note that this nonterminal appears only at the ends of bodies of E-productions. Thus, $\text{FOLLOW}(E')$ must be the same as $\text{FOLLOW}(E)$.

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

$$\text{FIRST}(E') = \{+, \epsilon\}$$

FOLLOW(T) = FOLLOW$(T') = \{+,), \$\}$. Notice that T appears in bodies only followed by E'. Thus, everything except ϵ that is in FIRST(E') must be in FOLLOW(T); that explains the symbol $+$. However, since FIRST(E') contains ϵ (i.e., $E' \stackrel{*}{\Rightarrow} \epsilon$), and E' is the entire string following T in the bodies of the E-productions, everything in FOLLOW(E) must also be in FOLLOW(T). That explains the symbols \$ and the right parenthesis. As for T', since it appears only at the ends of the T-productions, it must be that FOLLOW(T') = FOLLOW(T).

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

$\text{FOLLOW}(F) = \{+, *,), \$\}$. The reasoning is analogous to that for T in point (5).

Follow(F)=Follow(T)

Predictive parsing

Challenges:
At **each step** of a top-down parse, the key problem is that of **determining the production to be applied** for a nonterminal, say A.

(a) Recursive descent parsing: May require backtracking to find the correct A-production to be applied
(b) **Predictive parsing:** No backtracking!
looking ahead at the input a fixed number of symbols (**next symbols**) –
LL(k), LL(1) grammars

Predictive parsing

Parsing table M

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	$F \to \mathbf{id}$			$F \to (E)$		

LL(1) grammar => avoid confusion!!

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

First(α) and First(β) Disjoint sets

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 $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

ϵ is in FIRST(α).

then FIRST(β) and FOLLOW(A) are disjoint sets.

Basic concept of Predictive parsing

Input string w=abcd

One sentential form

S=> aXY….

Grammar productions

1. X-> **b**A…

First symbol

2. X-> bY……

Another sentential form

S=> aXb

Grammar productions

1. X-> €
2. X-> ……
3. **X->bY….**

We know that **b** **Follows** **X** in any sentential form …. **Follow(X)=b**

Parsing table M

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in $\text{FIRST}(A)$, add $A \to \alpha$ to $M[A, a]$.

Obvious

Input string w=bacd

One sentential form

S=> b**A**Y....

Grammar productions

1. A-> **a**X...

First symbol

2. A-> ......

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \to \alpha$ to $M[A, a]$.

2. If ϵ is in FIRST(α) then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

Input string w=abcd

One sentential form

S=> aAb

Grammar productions

1. A-> α=>€
2. X-> ……

We know that **b** **Follows** **A** in any sentential form …. **Follow(A)=b**

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \to \alpha$ to $M[A, a]$.

2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). $\square$

- production $E \to TE'$

$$\text{FIRST}(TE') = \text{FIRST}(T) = \{(, \textbf{id}\}$$

- Production $E' \to +TE'$

$$\text{FIRST}(+TE') = \{+\}$$

- $E' \to \epsilon$

$$\text{FOLLOW}(E') = \{), \$\}$$

$$\begin{array}{lcl} E & \to & T\ E' \\ E' & \to & +\ T\ E' \mid \epsilon \\ T & \to & F\ T' \\ T' & \to & *\ F\ T' \mid \epsilon \\ F & \to & (\ E\ ) \mid \textbf{id} \end{array}$$

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	$F \to \textbf{id}$			$F \to (E)$		

$T \rightarrow FT'$

First(FT')={(,id}

$T' \rightarrow *FT'$

First(*FT')={*}

$T' \rightarrow \epsilon$

Follow(T')={+,),$}

$$
\begin{array}{lll}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

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

$F \rightarrow (\ E\ ) \mid \mathbf{id}$

First((E))={(}

First(id)={id}

Example of Non-LL(1) grammar

- For every LL(1) grammar, **each parsing-table entry uniquely** identifies a production or signals an error.
- **left-recursive** or **ambiguous grammars** are not LL(1)

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

Input string
i b t i b t a e a

if b
then
if b
then
a
else
a

Example of Non-LL(1) grammar

Non - terminal	Input Symbol					
	a	b	e	i	t	$\$$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Predictive Parsing

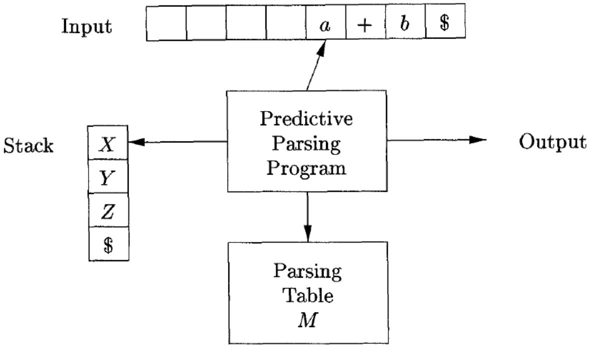
- **Non-recursive** version
  - maintaining a **stack explicitly**, rather than implicitly via recursive calls

INPUT: A string w and a parsing table M for grammar G.

OUTPUT: If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

Initial configuration

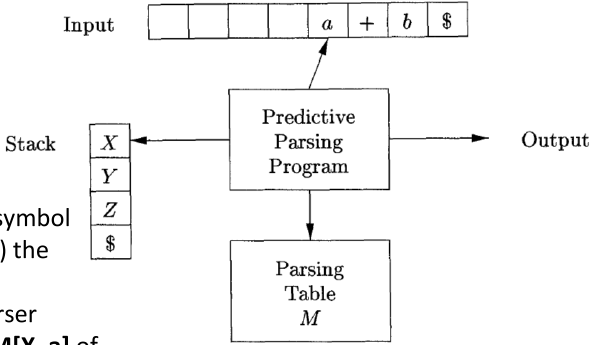
STACK	INPUT
$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$



Predictive Parsing

INPUT: A string w and a parsing table M for grammar G.

OUTPUT: If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

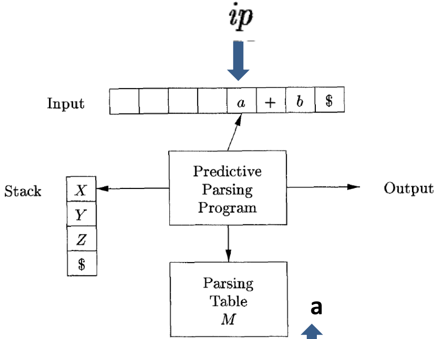


Initial configuration

STACK	INPUT
$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$

- The parser considers (i) the symbol on **top of the stack X**, and (ii) the current **input symbol a**.
- If **X is a nonterminal**, the parser chooses an X-production from **M[X, a]** of the parsing table.
- Otherwise, it checks for a **match** between the **terminal X** and current **input symbol a**.

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	$F \to \mathbf{id}$			$F \to (E)$		



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	$F \to \mathbf{id}$			$F \to (E)$		

a

set *ip* to point to the first symbol of w;
set X to the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
    if (X is a) pop the stack and advance *ip*;
    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;
    }
    set X to the top stack symbol;
}

$\mathbf{Y_1}$

$\mathbf{id} + \mathbf{id} * \mathbf{id}$

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	$F \to \mathbf{id}$			$F \to (E)$		

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 $E' \to \epsilon$

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 $E' \to \epsilon$

Leftmost derivation

$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} \mathbf{id}\, T'E' \underset{lm}{\Rightarrow} \mathbf{id}\, E' \underset{lm}{\Rightarrow} \mathbf{id} + TE' \underset{lm}{\Rightarrow} \cdots$$

Predictive Parsing

The stack contains a sequence of grammar symbols

If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols α such that

$$S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$$

$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} \mathbf{id}\, T'E' \underset{lm}{\Rightarrow} \mathbf{id}\, E' \underset{lm}{\Rightarrow} \underset{\substack{\uparrow \\ w}}{\boxed{\mathbf{id} +}}\, \overset{\substack{\alpha \\ \downarrow}}{T E'} \underset{lm}{\Rightarrow} \cdots$$