Language Processing Systems

Prof. Mohamed Hamada

Software Engineering Lab.
The University of Aizu
Japan

Syntax Analysis
(Parsing)

Some Basic Definitions

syntax: the way in which words are put together to form phrases, clauses, or sentences. The rules governing the formation of statements in a programming language.

syntax analysis: the task concerned with fitting a sequence of tokens into a specified syntax.

parsing: To break a sentence down into its component parts of speech with an explanation of the form, function, and syntactical relationship of each part.

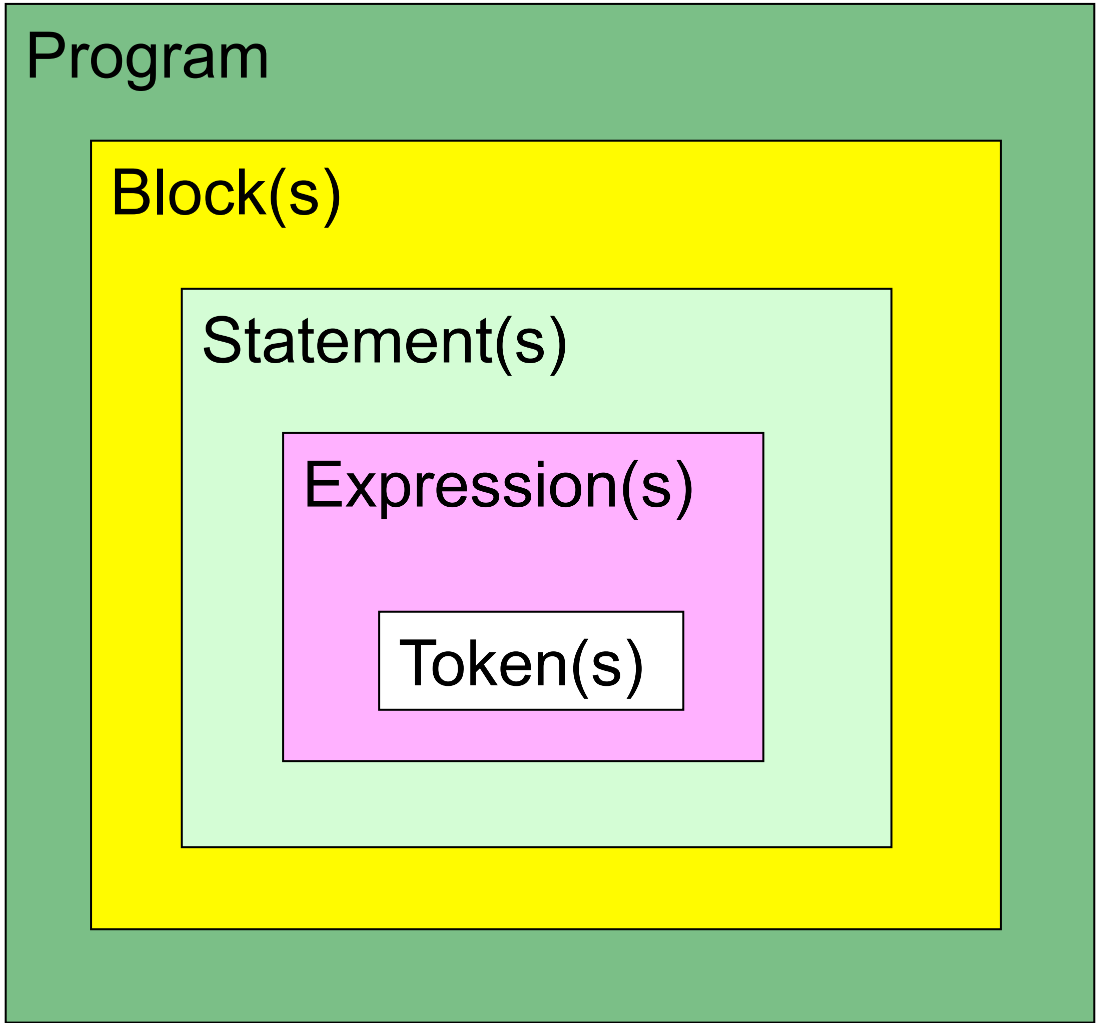
Some Basic Definitions

parsing = lexical analysis + syntax analysis

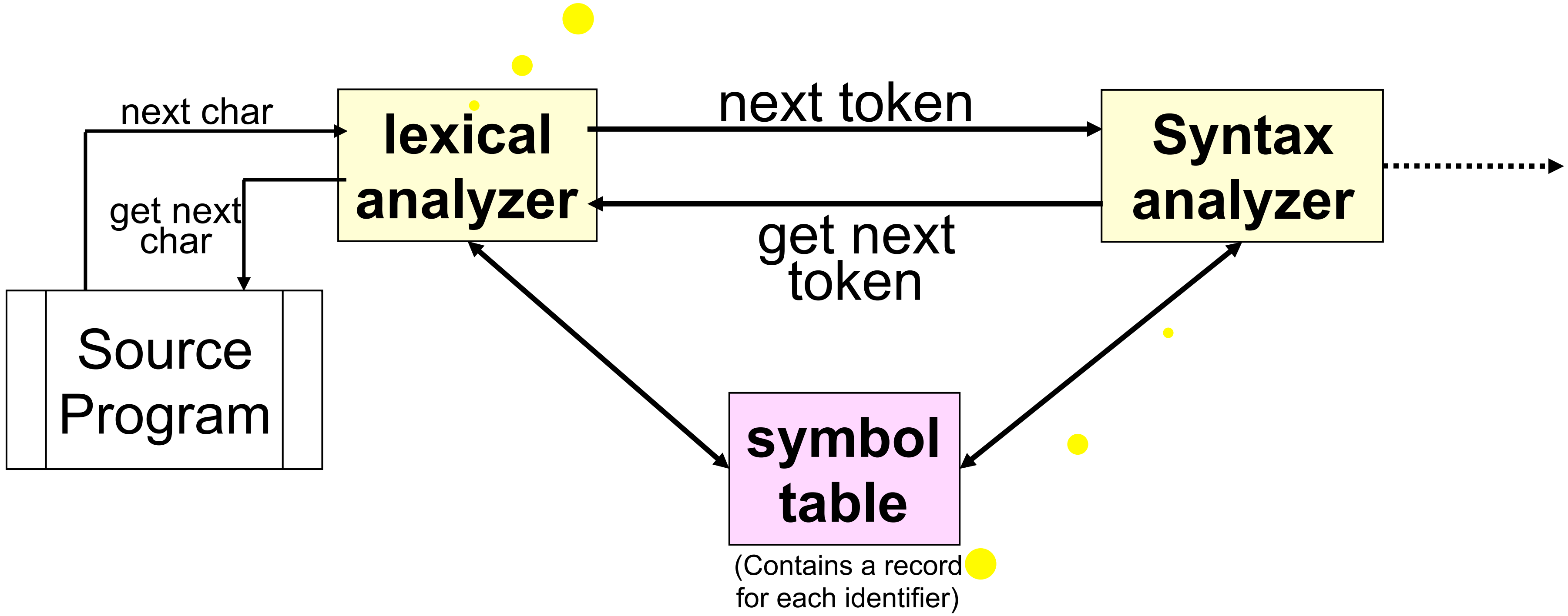
semantic analysis: the task concerned with calculating the program's meaning.

Some Basic Definitions

Syntactic structure: the syntactic structure of programming languages can be informally expressed by the following diagram.



1. Uses Regular Expressions to define tokens
2. Uses Finite Automata to recognize tokens



Uses Top-down parsing or Bottom-up parsing

To construct a Parse tree

Syntax errors

Parsing errors include:

1. misspelling of identifier, keyword, or operator

2. arithmetic expression with unbalanced parentheses

3. punctuation errors such as using comma in place of semicolon

4. missing brackets, semicolons, etc.

Error recovery

The error handler in a parser has the following jobs:

1. report the presence of errors clearly and accurately

2. quick recovery of errors

3. not to slow the processing of programs

Example:

The following C code shows some examples of syntax errors:

```
#include<stdio.h>
int max(int I;int j)
{
if(i>j) return(i)
return(j);
}
void main()
{
int x, y,
scanf("%d %d", x, y);
printf("%d", max(x,y) ;
}
```

Example:

A typical compilation of this erroneous program gives the following list of errors:

1. error C2235: ';' in formal parameter list
2. error C2059: syntax error : ')'
3. error C2239: unexpected token 'f' following declaration of 'j'
4. error C2078: too many initializers
5. error C2660: 'max' : function does not take 2 parameters
6. error C2143: syntax error : missing ')' before ';'

Example:

The correct version of this program is

```
#include<stdio.h>
int max(int i, int j)
{
if(i>j) return(i);
return(j);
}
void main()
{
int x, y;
scanf("%d %d", x, y);
printf("%d", max(x,y));
}
```

Definition of Context-Free Grammars

A context-free grammar $G = (T, N, S, P)$ consists of:

1. T, a set of *terminals* (scanner tokens).
2. N, a set of *nonterminals* (syntactic variables generated by productions).
3. S, a designated *start* nonterminal.
4. P, a set of *productions*. Each production has the form, $A ::= \alpha$, where A is a nonterminal and α is a *sentential form* , i.e., a string of zero or more grammar symbols (terminals/nonterminals).

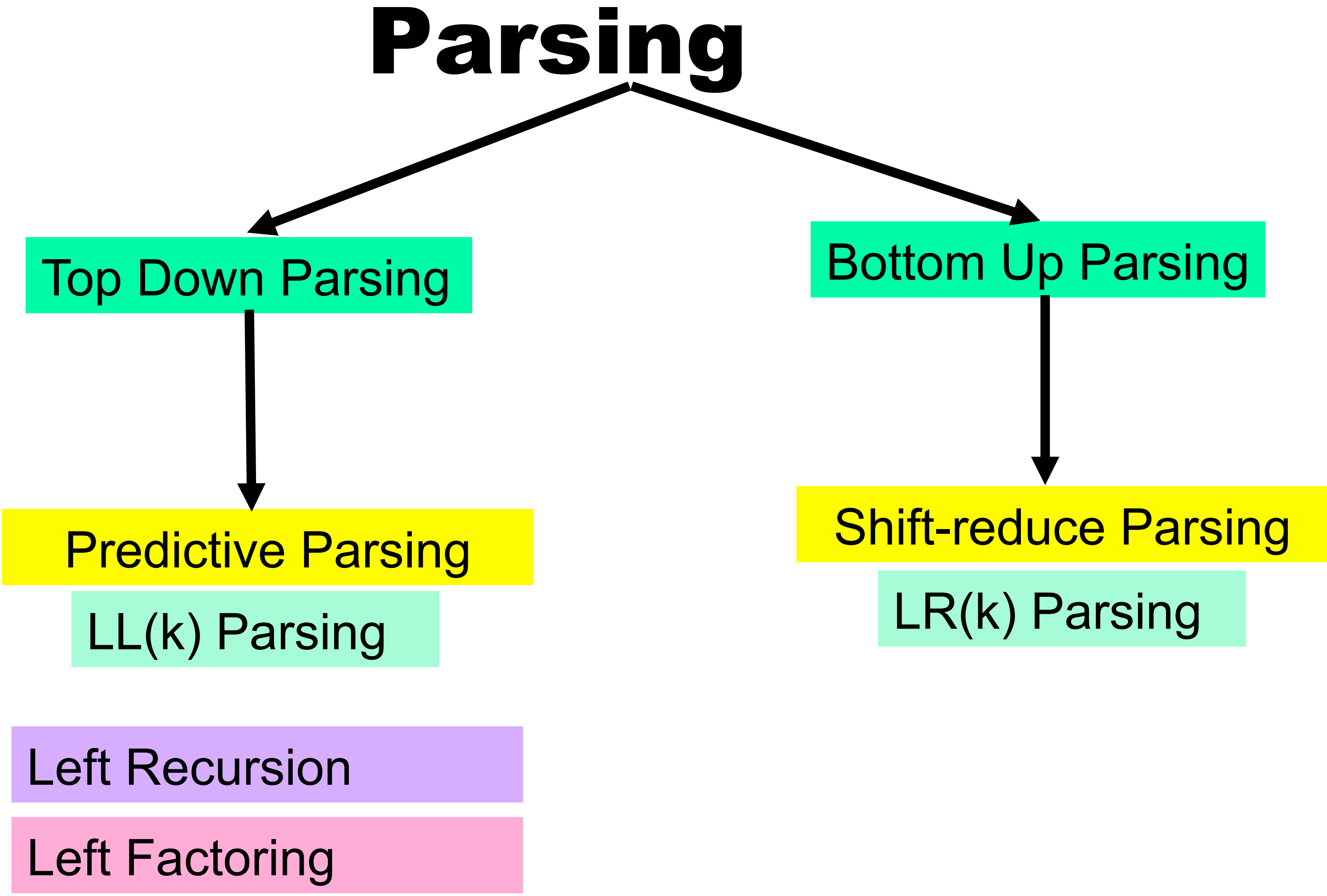
Definition of Context-Free Grammars

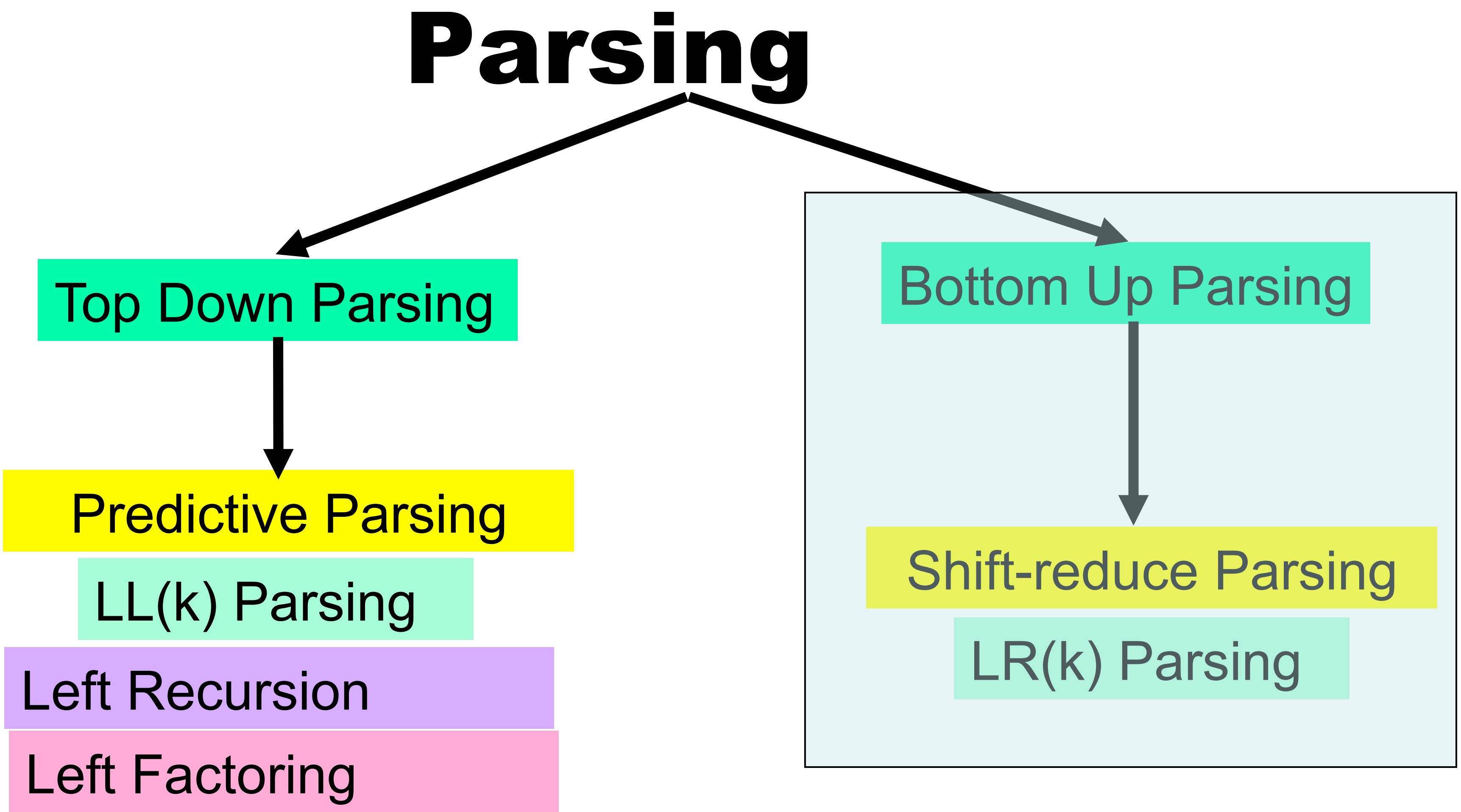
Grammars offer several significant advantages:

1. Easy to understand and construct programs
2. Easy parsing
3. Easy error detection and handling
4. Easy language extension

Syntax Analysis

Syntax Analysis Problem Statement: To find a derivation sequence in a grammar ***G*** for the input token stream (or say that none exists).


Parsing
Top Down Parsing
Bottom Up Parsing
Predictive Parsing
LL(k) Parsing
Shift-reduce Parsing
LR(k) Parsing
Left Recursion
Left Factoring



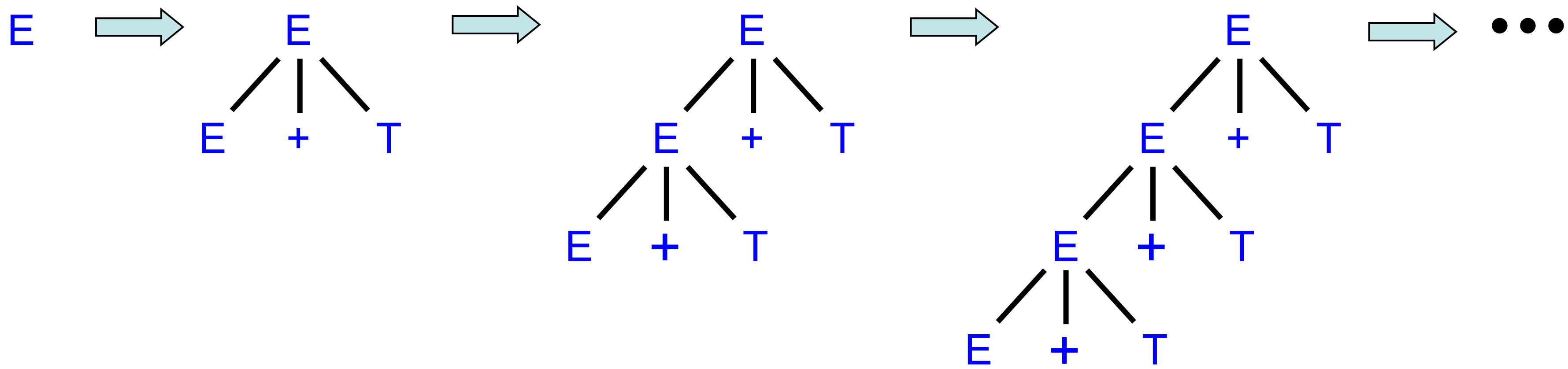
Top-down parsers: starts constructing the parse tree at the top (root) of the tree and move down towards the leaves.
Easy to implement by hand, but work with restricted grammars.
Example: predictive parsers

Left Recursion

Consider the grammar:

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

A top-down parser might loop forever when parsing an expression using this grammar



Left Recursion

Consider the grammar:

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

A grammar that has at least one production of the form $A \Rightarrow A\alpha$ is a left recursive grammar.

Top-down parsers do not work with left-recursive grammars.

Left-recursion can often be eliminated by rewriting the grammar.

Left Recursion

This left-recursive grammar:

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

Can be re-written to eliminate the immediate left recursion:

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

Predictive Parsing

Consider the grammar:

stm → **if** expr **then** stmt **else** stmt
  | **while** expr **do** stmt
  | **begin** stmt_list **end**

A parser for this grammar can be written with the following simple structure:

```
switch(gettoken())
{
        case if:
                ....
                break;
        case while:
                ....
                break;
        case begin:
                ....
                break;
        default:
                reject input;
}
```

Based only on the first token, the parser knows which rule to use to derive a statement.

Therefore this is called a predictive parser.

Left Factoring

The following grammar:

stmt → **if** expr **then** stmt **else** stmt
| **if** expr **then** stmt

Cannot be parsed by a predictive parser that looks one element ahead.

But the grammar can be re-written:

stmt → **if** expr **then** stmt stmt'
stmt'→ **else** stmt | λ

Where λ is the empty string.

Rewriting a grammar to eliminate multiple productions starting with the same token is called left factoring.

Left Factoring

The basic idea is, in general, as follows:

1. let $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ be two production rules for the nonterminal symbol A
2. if the input begins with a nonempty string derived from α
3. and we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$
4. then we may defer the decision by expanding A to $\alpha A'$
5. after seeing the input derived from α, we expand A' to β_1 or to β_2
6. this means, left-factored, the original productions become

$$A \rightarrow \alpha A'$$

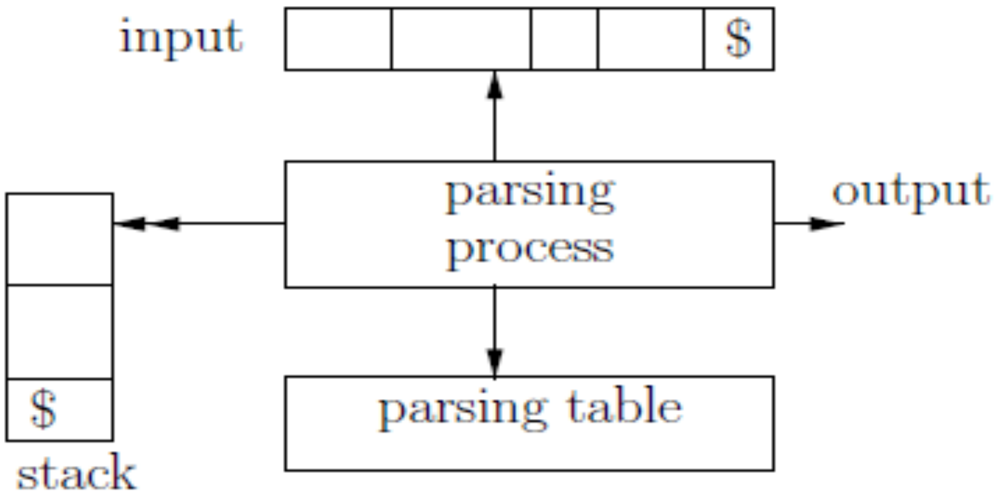
$$A' \rightarrow \beta_1 \mid \beta_2$$

A Predictive Parser

How it works?

1. Construct the *parsing table* from the given grammar

2. Apply the predictive parsing algorithm to construct the parse tree



A Predictive Parser

1. Construct the *parsing table* from the given grammar

The following algorithm shows how we can construct the *parsing table*:

Input: a grammar G

Output: the corresponding parsing table M

Method: For each production A → α of the grammar do the following steps:

1. For each terminal a in FIRST(α), add A → α to M[A,a].
2. If λ in FIRST(α), add A → α to M[A,b] for each terminal b in FOLLOW(A).
3. If λ FIRST(α) and $ in FOLLOW(A), add A → α to M[A,$]

How to construct FIRST and FOLLOW operations?

The Parsing Table

How to construct FIRST and FOLLOW operations?

Example

Given this grammar:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \lambda$
$F \rightarrow (E) \mid \mathbf{id}$

How is this parsing table built?

PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E′		$E' \rightarrow +TE'$			$E' \rightarrow \lambda$	$E' \rightarrow \lambda$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T′		$T' \rightarrow \lambda$	$T' \rightarrow *FT'$		$T' \rightarrow \lambda$	$T' \rightarrow \lambda$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

FIRST and FOLLOW

We need to build a FIRST set and a FOLLOW set for each symbol in the grammar.

The elements of FIRST and FOLLOW are terminal symbols.

FIRST(α) is the set of terminal symbols that can begin any string derived from α.

FOLLOW(α) is the set of terminal symbols that can follow α:

$$t \in \text{FOLLOW}(\alpha) \leftrightarrow \exists \text{ derivation containing } \alpha t$$

Rules to Create FIRST

GRAMMAR:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \lambda$
$F \rightarrow (E) \mid \textbf{id}$

SETS:

FIRST(**id**) = {**id**}
FIRST(*) = {*}
FIRST(**+**) = {**+**}
FIRST(**(**) = {**(**}
FIRST(**)**) = {**)**}
FIRST(E') = $\cancel{\{\varepsilon\}}$ {**+**, λ}
FIRST(T') = $\cancel{\{\varepsilon\}}$ {*, λ}
FIRST(F) = {(, id}
FIRST(T) = FIRST(F) = {(, id}
FIRST(E) = FIRST(T) = {(, id}

FIRST rules:

1. If X is a <u>terminal</u>, FIRST(X) = {X}
2. If $X \rightarrow \lambda$, then $\varepsilon \in$ FIRST(X)
3. If $X \rightarrow Y_1 Y_2 \cdots Y_k$
   and $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \lambda$
   and $a \in$ FIRST(Y_i)
   then $a \in$ FIRST(X)

… to Create FOLLOW

FIRST(E’) = {+, λ}
FIRST(T’) = {* , λ}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

GRAMMAR:

E → TE’
E’ → +TE’ | λ
T → FT’
T’ → *FT’ | λ
F → (E) | **id**

SETS:

FOLLOW(E) = ~~{$}~~ {), $}
FOLLOW(E’) = { **)**, **$**}
FOLLOW(T) = { **)**, **$**}

FOLLOW rules:

1. If S is the start symbol, then $\$ \in \text{FOLLOW}(S)$
2. If $A \rightarrow \alpha B \beta$,
   and $a \in \text{FIRST}(\beta)$
   and $a \neq \lambda$
   then $a \in \text{FOLLOW}(B)$
3. If $A \rightarrow \alpha B$
   and $a \in \text{FOLLOW}(A)$
   then $a \in \text{FOLLOW}(B)$

3a. If $A \rightarrow \alpha B \beta$
   and $\beta \overset{*}{\Rightarrow} \lambda$
   and $a \in \text{FOLLOW}(A)$
   then $a \in \text{FOLLOW}(B)$

A and B are non-terminals,
α and β are strings of grammar symbols

FIRST(E’) = {+, λ}
FIRST(T’) = {* , λ}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

F to Create FOLLOW

GRAMMAR:

E → TE’
E’ → +TE’ | λ
T → FT’
T’ → *FT’ | λ
F → (E) | **id**

SETS:

FOLLOW(E) = {**)**, **$**}

FOLLOW(E’) = { **)**, **$**}

FOLLOW(T) = ~~{), $}~~ {**+**, **)**, **$**}

FOLLOW rules:

1. If S is the start symbol, then $\$ \in FOLLOW(S)$
2. If $A \rightarrow \alpha B \beta$,
   and $a \in FIRST(\beta)$
   and $a \neq \lambda$
   then $a \in FOLLOW(B)$
3. If $A \rightarrow \alpha B$
   and $a \in FOLLOW(A)$
   then $a \in FOLLOW(B)$

3a. If $A \rightarrow \alpha B \beta$
   and $\beta \overset{*}{\Rightarrow} \lambda$
   and $a \in FOLLOW(A)$
   then $a \in FOLLOW(B)$

A and B are non-terminals,
α and β are strings of grammar symbols

FIRST(E’) = {**+**, λ}
FIRST(T’) = {* , λ}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

to Create FOLLOW

GRAMMAR:

E → TE’
E’ → +TE’ | λ
T → FT’
T’ → *FT’ | λ
F → (E) | **id**

SETS:

FOLLOW(E) = {**)**, **$**}

FOLLOW(E’) = { **)**, **$**}

FOLLOW(T) = {**+**, **)**, **$**}

FOLLOW(T’) = {**+**, **)**, **$**}

FOLLOW rules:

1. If S is the start symbol, then $\$ \in \text{FOLLOW}(S)$
2. If $A \rightarrow \alpha B \beta$,
   and $a \in \text{FIRST}(\beta)$
   and $a \neq \lambda$
   then $a \in \text{FOLLOW}(B)$
3. If $A \rightarrow \alpha B$
   and $a \in \text{FOLLOW}(A)$
   then $a \in \text{FOLLOW}(B)$

3a. If $A \rightarrow \alpha B \beta$
   and $\beta \overset{*}{\Rightarrow} \lambda$
   and $a \in \text{FOLLOW}(A)$
   then $a \in \text{FOLLOW}(B)$

A and B are non-terminals,
α and β are strings of grammar symbols

F... to Create FOLLOW

FIRST(E') = {+, λ}
FIRST(T') = {* , λ}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

GRAMMAR:

$E \to TE'$
$E' \to +TE' \mid \lambda$
$T \to FT'$
$T' \to *FT' \mid \lambda$
$F \to (E) \mid$ **id**

SETS:

FOLLOW(E) = {**)**, **$**}

FOLLOW(E') = { **)**, **$**}

FOLLOW(T) = {**+**, **)**, **$**}

FOLLOW(T') = {**+**, **)**, **$**}

FOLLOW(F) = {**+**, **)**, **$**}

FOLLOW rules:

1. If S is the start symbol, then $\$ \in FOLLOW(S)$
2. If $A \to \alpha B \beta$,
   and $a \in FIRST(\beta)$
   and $a \neq \lambda$
   then $a \in FOLLOW(B)$
3. If $A \to \alpha B$
   and $a \in FOLLOW(A)$
   then $a \in FOLLOW(B)$

3a. If $A \to \alpha B \beta$
   and $\beta \overset{*}{\Rightarrow} \lambda$
   and $a \in FOLLOW(A)$
   then $a \in FOLLOW(B)$

A and B are non-terminals,
α and β are strings of grammar symbols

FIRST(E') = {**+**, λ}

FIRST(T') = {* , λ}

FIRST(F) = {(, id}

FIRST(T) = {(, id}

FIRST(E) = {(, id}

... to Create FOLLOW

GRAMMAR:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \lambda$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \lambda$

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

SETS:

FOLLOW(E) = {**)**, **$**}

FOLLOW(E') = { **)**, **$**}

FOLLOW(T) = {**+**, **)**, **$**}

FOLLOW(T') = {**+**, **)**, **$**}

FOLLOW(F) = ~~{**+**, **)**, **$**}~~ {**+**, *, **)**, **$**}

FOLLOW rules:

1. If S is the start symbol, then $\$ \in FOLLOW(S)$
2. If $A \rightarrow \alpha B\beta$,
   and $a \in FIRST(\beta)$
   and $a \neq \lambda$
   then $a \in FOLLOW(B)$
3. If $A \rightarrow \alpha B$
   and $a \in FOLLOW(A)$
   then $a \in FOLLOW(B)$

3a. If $A \rightarrow \alpha B\beta$
   and $\beta \overset{*}{\Rightarrow} \lambda$
   and $a \in FOLLOW(A)$
   then $a \in FOLLOW(B)$

A and B are non-terminals,
α and β are strings of grammar symbols

GRAMMAR:

$E \to TE'$
$E' \to +TE' \mid \lambda$
$T \to FT'$
$T' \to *FT' \mid \lambda$
$F \to (E) \mid$ **id**

FIRST SETS:

FIRST(E') = {**+**, λ}
FIRST(T') = {* , λ}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

FOLLOW SETS:

FOLLOW(E) = {**)**, **$**}
FOLLOW(E') = { **)**, **$**}
FOLLOW(T) = {**+**, **)**, **$**}
FOLLOW(T') = {**+**, **)**, **$**}
FOLLOW(F) = {**+**, *****, **)**, **$**}

1. If $A \to \alpha$:
   if $a \in \text{FIRST}(\alpha)$, add $A \to \alpha$ to $M[A, a]$

PARSING TABLE:

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \lambda$	$E' \to \lambda$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \lambda$	$T' \to *FT'$		$T' \to \lambda$	$T' \to \lambda$
F	$F \to$ **id**			$F \to (E)$		

GRAMMAR:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \lambda$
$F \rightarrow (E) \mid \textbf{id}$

FIRST SETS:

FIRST(E') = {**+**, λ}
FIRST(T') = {* , λ}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

FOLLOW SETS:

FOLLOW(E) = {**)**, **$**}
FOLLOW(E') = { **)**, **$**}
FOLLOW(T) = {**+**, **)**, **$**}
FOLLOW(T') = {**+**, **)**, **$**}
FOLLOW(F) = {**+**, *****, **)**, **$**}

1. If $A \rightarrow \alpha$:
   if $a \in \text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$

PARSING TABLE M:

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

GRAMMAR:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \lambda$
$F \rightarrow (\ E\ ) \mid \textbf{id}$

FIRST SETS:

FIRST(E') = {**+**, λ}
FIRST(T') = {*, λ}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

FOLLOW SETS:

FOLLOW(E) = {**)**, **$**}
FOLLOW(E') = { **)**, **$**}
FOLLOW(T) = {**+**, **)**, **$**}
FOLLOW(T') = {**+**, **)**, **$**}
FOLLOW(F) = {**+**, *, **)**, **$**}

1. If $A \rightarrow \alpha$:
   if $a \in \text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$

PARSING TABLE M:

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

GRAMMAR:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \lambda$
$F \rightarrow (E) \mid \mathbf{id}$

FIRST SETS:

FIRST(E') = {**+**, λ}
FIRST(T') = {* , λ}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

FOLLOW SETS:

FOLLOW(E) = {**)**, **$**}
FOLLOW(E') = { **)**, **$**}
FOLLOW(T) = {**+**, **)**, **$**}
FOLLOW(T') = {**+**, **)**, **$**}
FOLLOW(F) = {**+**, *, **)**, **$**}

1. If $A \rightarrow \alpha$:
   if $a \in$ FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$

PARSING TABLE M:

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

GRAMMAR:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \lambda$
$F \rightarrow (E) \mid \mathbf{id}$

FIRST SETS:

FIRST(E') = {+, λ}
FIRST(T') = {* , λ}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

FOLLOW SETS:

FOLLOW(E) = {), $}
FOLLOW(E') = {), $}
FOLLOW(T) = {+,), $}
FOLLOW(T') = {+,), $}
FOLLOW(F) = {+, *,), $}

1. If $A \rightarrow \alpha$:
   if $a \in \text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$

PARSING TABLE M:

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \lambda$	$E' \rightarrow \lambda$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \lambda$	$T' \rightarrow *FT'$		$T' \rightarrow \lambda$	$T' \rightarrow \lambda$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

GRAMMAR:

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

FIRST SETS:

FIRST(E') = {**+**, λ}
FIRST(T') = {* , λ}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

FOLLOW SETS:

FOLLOW(E) = {**)**, **$**}
FOLLOW(E') = { **)**, **$**}
FOLLOW(T) = {**+**, **)**, **$**}
FOLLOW(T') = {**+**, **)**, **$**}
FOLLOW(F) = {**+**, *****, **)**, **$**}

1. If $A \to \alpha$:
   if $a \in$ FIRST(α), add $A \to \alpha$ to M[A, a]
2. If $A \to \alpha$:
   if $\lambda \in$ FIRST(α), add $A \to \alpha$ to M[A, b]
   for each terminal $b \in$ FOLLOW(A),

PARSING TABLE M:

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

GRAMMAR:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \lambda$
$F \rightarrow (E) \mid \textbf{id}$

FIRST SETS:

FIRST(E') = {+, λ}
FIRST(T') = {* , λ}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

FOLLOW SETS:

FOLLOW(E) = {), $}
FOLLOW(E') = {), $}
FOLLOW(T) = {+,), $}
FOLLOW(T') = {+,), $}
FOLLOW(F) = {+, *,), $}

1. If $A \rightarrow \alpha$:
   if $a \in \text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
2. If $A \rightarrow \alpha$:
   if $\lambda \in \text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, b]$
   for each terminal $b \in \text{FOLLOW}(A)$,

PARSING TABLE M:

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

GRAMMAR:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \lambda$
$F \rightarrow (E) \mid$ **id**

FIRST SETS:

FIRST(E') = {**+**, λ}
FIRST(T') = {* , λ}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

FOLLOW SETS:

FOLLOW(E) = {**)**, **$**}
FOLLOW(E') = { **)**, **$**}
FOLLOW(T) = {**+**, **)**, **$**}
FOLLOW(T') = {**+**, **)**, **$**}
FOLLOW(F) = {**+**, *****, **)**, **$**}

1. If $A \rightarrow \alpha$:
   if $a \in \mathrm{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
2. If $A \rightarrow \alpha$:
   if $\lambda \in \mathrm{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, b]$
   for each terminal $b \in \mathrm{FOLLOW}(A)$,
3. If $A \rightarrow \alpha$:
   if $\lambda \in \mathrm{FIRST}(\alpha)$, and $\$ \in \mathrm{FOLLOW}(A)$,
   add $A \rightarrow \alpha$ to $M[A, \$]$

PARSING TABLE M:

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E → TE′			E → TE′		
E′		E′ → +TE′			E′ → λ	E′ → λ
T	T → FT′			T → FT′		
T′		T′→ λ	T′ → *FT′		T′ → λ	T′ → λ
F	F → **id**			F → (E)		