BCSE307L – COMPILER DESIGN

TEXT BOOK:

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques and Tools", Second Edition, Pearson Education Limited, 2014.

Module:2	SYNTAX ANALYSIS	8 hours
Role of Parser- Parse Tree - Elimination of Ambiguity – Top Down Parsing - Recursive Descent Parsing - LL (1) Grammars – Shift Reduce Parsers- Operator Precedence Parsing - LR Parsers, Construction of SLR Parser Tables and Parsing- CLR Parsing- LALR Parsing.		

Syntax analysis

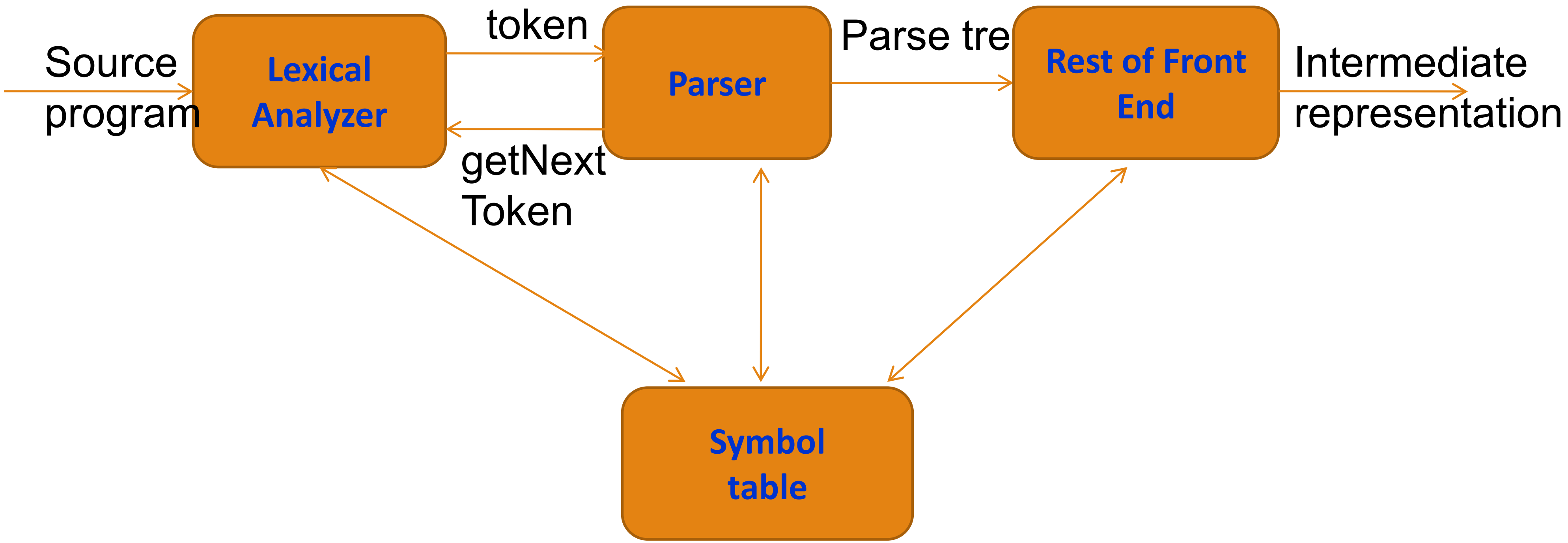
Introduction to Syntax Analysis

CFG

Types of Parser

The role of parser

- Parser works on a stream of tokens.
- The smallest item is a token.



Syntax Analyzer

Syntax Analyzer creates the syntactic structure of the given source program.

This syntactic structure is mostly a *parse tree*.

Syntax Analyzer is also known as *parser*.

The syntax of a programming is described by a *context-free grammar (CFG)*.

Syntax Analyzer

The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.

- If it satisfies, the parser creates the parse tree of that program.
- Otherwise the parser gives the error messages.

A context-free grammar

- gives a precise syntactic specification of a programming language.
- the design of the grammar is an initial phase of the design of a compiler.
- a grammar can be directly converted into a parser by some tools.

Expression Grammar

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

Parsers

We categorize the parsers into two groups:

1. **Top-Down Parser**
   - the parse tree is created top to bottom, starting from the root.
2. **Bottom-Up Parser**
   - the parse is created bottom to top; starting from the leaves

Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).

Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.

- LL for top-down parsing
- LR for bottom-up parsing

Syntax Tree

A parse tree is a record of the rules (and tokens) used to match some input text

A syntax tree records the structure of the input and is insensitive to the grammar that produced it.

Syntax tree

Parse tree: interior nodes are non-terminals, leaves are terminals
Syntax tree: interior nodes are "operators", leaves are operands

Parse tree: rarely constructed as a data structure
Syntax tree: when representing a program in a tree structure usually use a syntax tree

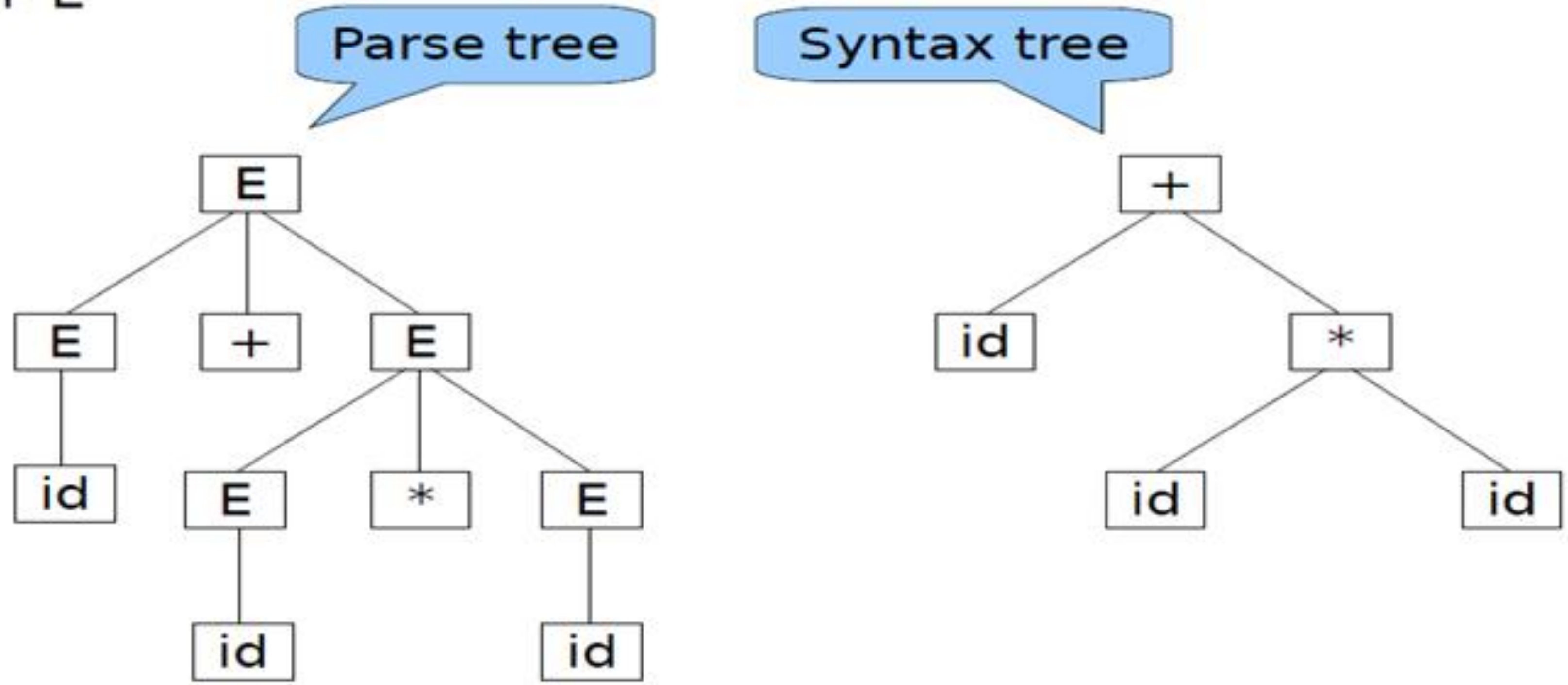
Parse tree: Represents the *concrete syntax* of a program
Syntax tree: Represents the *abstract syntax* of a program (the semantics)

A syntax tree is often called *abstract syntax tree* or **AST**

Syntax and parse tree examples

Grammar:
E → E * E
| E + E
| id

Program: a + b * c



CONTEXT FREE GRAMMAR (CFG)

- The formal definition of a context-free grammar
- Notational Conventions
- Derivations
- Parse Tree and Derivations
- Ambiguity
- Verifying the language generated by a Grammar
- Context-free grammar versus regular expression

Context-Free Grammars

Inherently recursive structures of a programming language are defined by a context-free grammar.

In a context-free grammar G = (V,T,P,S), we have:

- **T -** A finite set of **terminals** (in our case, this will be the set of tokens)
- **V -** A finite set of **non-terminals** (syntactic-variables)
- **P -** A finite set of **productions** rules in the following form
  - $A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string)
- **S -** A **start symbol** (one of the non-terminal symbol)

Notational Conventions

1. These symbols are terminals:
   (a) Lowercase letters early in the alphabet, such as a, b, c.
   (b) Operator symbols such as $+$, $*$, and so on.
   (c) Punctuation symbols such as parentheses, comma, and so on.
   (d) The digits $0, 1, \ldots, 9$.
   (e) Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.
2. These symbols are nonterminals:
   (a) Uppercase letters early in the alphabet, such as A, B, C.
   (b) The letter S, which, when it appears, is usually the start symbol.
   (c) Lowercase, italic names such as *expr* or *stmt*.
   (d) When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs. For example, nonterminals for expressions, terms, and factors are often represented by E, T, and F, respectively.

Notational Conventions

3. Uppercase letters late in the alphabet, such as X, Y, Z, represent *grammar symbols*; that is, either nonterminals or terminals.

4. Lowercase letters late in the alphabet, chiefly $u, v, \ldots, z$, represent (possibly empty) strings of terminals.

5. Lowercase Greek letters, α, β, γ for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as $A \rightarrow \alpha$, where A is the head and α the body.

6. A set of productions $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2, \ldots,$ $A \rightarrow \alpha_k$ with a common head A (call them A-*productions*), may be written $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$. Call $\alpha_1, \alpha_2, \ldots, \alpha_k$ the *alternatives* for A.

7. Unless stated otherwise, the head of the first production is the start symbol.

Example

Using these conventions, the grammar of Example

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

The notational conventions tell us that E, T, and F are nonterminals, with E the start symbol. The remaining symbols are terminals.

Expression Grammar Example:

$E \rightarrow E + E \;|\; E - E \;|\; E * E \;|\; E / E \;|\; -E$

$E \rightarrow (E)$

$E \rightarrow id$

CFG - Terminology

L(G) is *the language of G* (the language generated by G) which is a set of sentences.

A sentence of L(G) is a string of terminal symbols of G.

If S is the start symbol of G then

ω is a sentence of L(G) iff $S \stackrel{+}{\Rightarrow} \omega$ where ω is a string of terminals of G.

If G is a context-free grammar, L(G) is a *context-free language*.

Two grammars are *equivalent* if they produce the same language.

$S \stackrel{*}{\Rightarrow} \alpha$

- If α contains non-terminals, it is called as a *sentential* form of G.
- If α does not contain non-terminals, it is called as a *sentence* of G.

Derivations

A sequence of replacements of non-terminal symbols is called a **derivation** of id+id from E.

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

$\Rightarrow$ * : derives in one step

$\Rightarrow$ $^{+}$: derives in zero or more steps

$\Rightarrow$: derives in one or more steps

Derivations

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

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

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

1. $\alpha \overset{*}{\Rightarrow} \alpha$, for any string α, and

2. If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$.

Derivations

$E \Rightarrow E+E$

E+E derives from E

- we can replace E by E+E
- to able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar.

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$

Derivation Example

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

OR

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.

If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.

If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

Left-Most and Right-Most Derivations

Left-Most Derivation

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E) \underset{lm}{\Rightarrow} -(id+id)$$

Right-Most Derivation

$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+id) \underset{rm}{\Rightarrow} -(id+id)$$

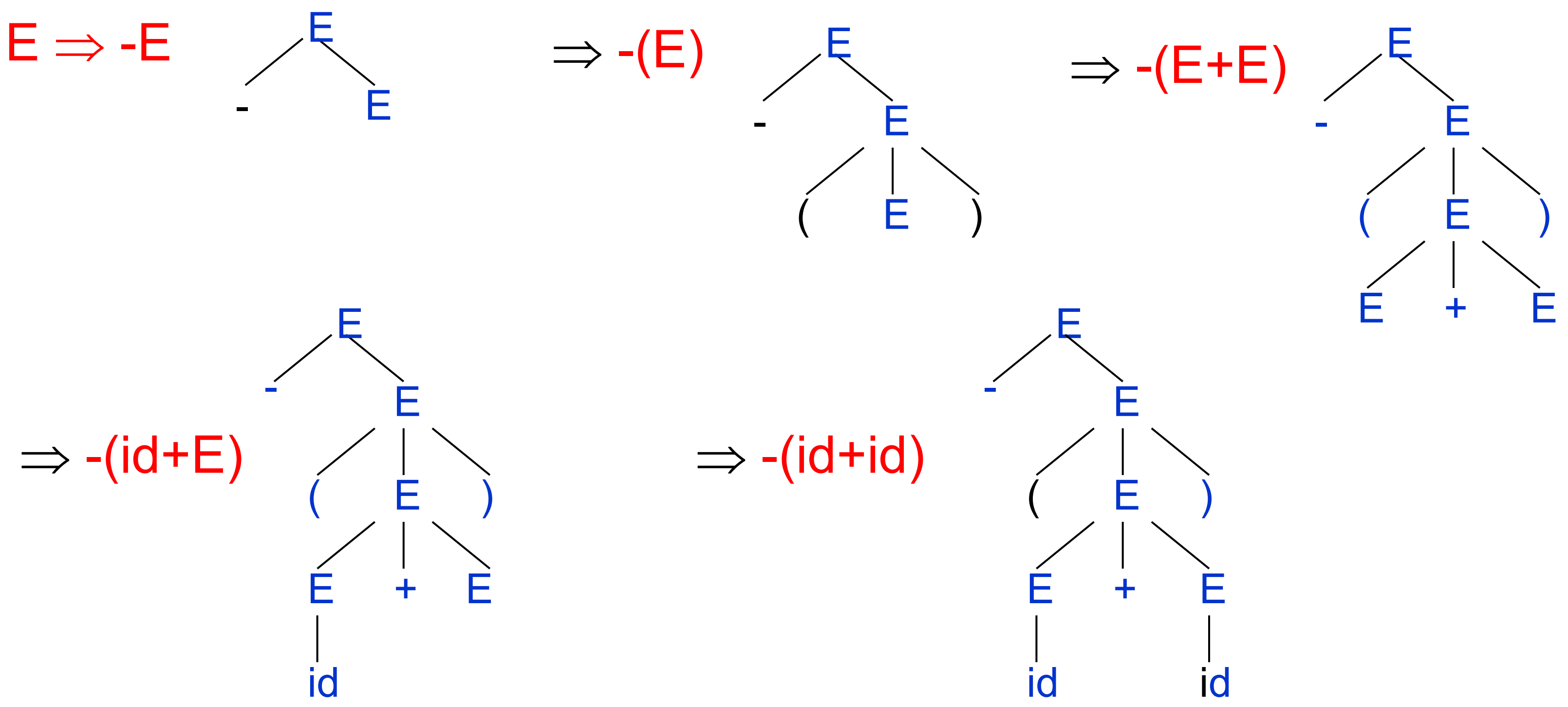
We will see that the top-down parsers try to find the left-most derivation of the given source program.

We will see that the bottom-up parsers try to find the right-most derivation of the given source program in the reverse order.

Parse Tree

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.

- A parse tree can be seen as a graphical representation of a derivation.



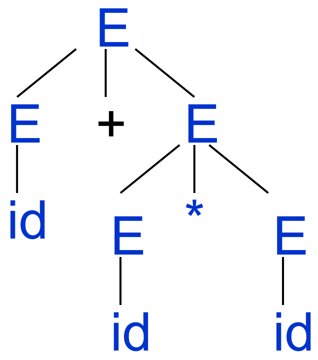
Ambiguity

- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.

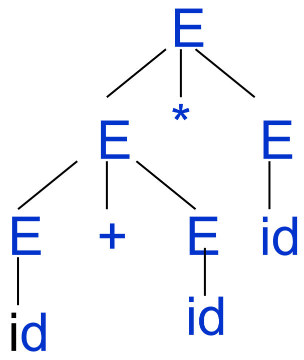
Ambiguous:

Two Different Parse Tree:

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



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



Ambiguity

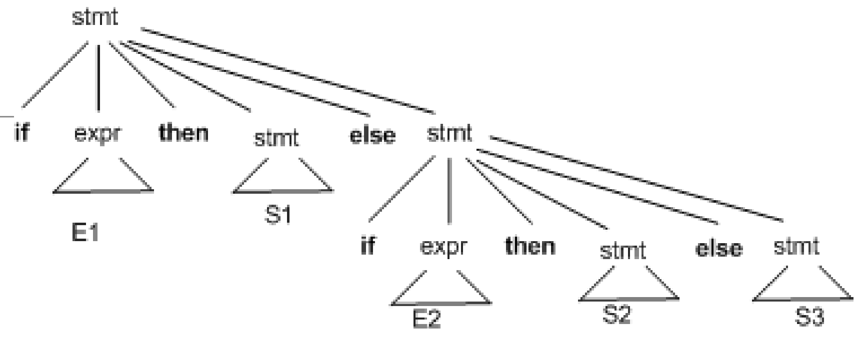
stmt $\longrightarrow$ **If** expr **then** stmt

| **If** expr **then** stmt **else** stmt

| **other**

One Parse Tree:

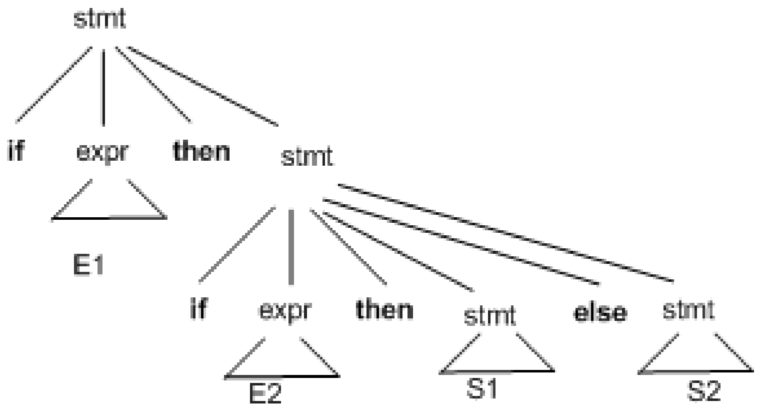
if E_1 then S1 else if E_2 then S2 else S3

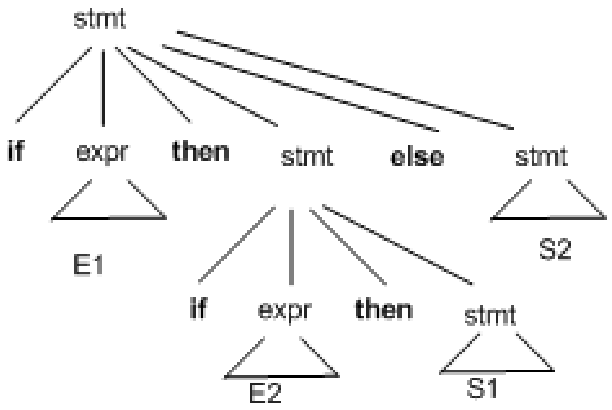


Ambiguous:

Two Different Parse Tree:

if E_1 then if E_2 then S_1 else S_2





Ambiguity

For the most parsers, the grammar must be unambiguous.

---

unambiguous grammar

→ unique selection of the parse tree for a sentence

We should eliminate the ambiguity in the grammar during the design phase of the compiler.

An unambiguous grammar should be written to eliminate the ambiguity.

We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.

Verifying the language generated by a Grammar

Expression $S \rightarrow (S)\ S \mid \varepsilon$

$$S \underset{lm}{\Rightarrow} (S)\ S \underset{lm}{\Rightarrow} (x)S \underset{lm}{\Rightarrow} (x)y$$

Checking the string (x)y is balanced with equal number of right and left parentheses and every prefix has at least as many left parentheses as right

$w \Rightarrow (x)y$ is also derivable form S.

Context-free grammar versus regular expression

R.E = (a|b)*abb

CFG:

S → aS | bS | aA

A → bB

B → bC

C → Ɛ

Writing a Grammar

- Lexical versus Syntactic Analysis
- Eliminating Ambiguity
- Elimination of Left Recursion
- Elimination of left Factoring

Elimination of ambiguity

Idea:

- A statement appearing between a **then** and an **else** must be matched (Match each **else** with the closest previous unmatched **then**)

stmt $\longrightarrow$ matched_stmt

| open_stmt

matched_stmt $\longrightarrow$ If expr **then** matched_stmt **else** matched_stmt

| **other**

open_stmt $\longrightarrow$ If expr **then** stmt

| If expr **then** matched_stmt **else** open_stmt

Ambiguity

- We prefer the first parse tree (else matches with closest if).
- So, we have to disambiguate our grammar to reflect this choice.
- The unambiguous grammar will be:

stmt → matched_stmt | unmatched_stmt

matchedstmt → `if` expr `then` matched_stmt `else` matchedstmt |
otherstmts

unmatchedstmt → `if` expr `then` stmt |
`if` expr `then` matched_stmt `else` unmatched_stmt

Expression Grammar Example:

$$E \rightarrow E + E \;|\; E - E \;|\; E * E \;|\; E / E \;|\; - E$$

$$E \rightarrow (E)$$

$$E \rightarrow id$$

Simple Arithmetic Expression (Unambiguous Expression) Grammar

expr → expr + term

expr → expr - term

expr → term

term → term * factor

term → term / factor

term → factor

factor → (expr)

factor → id

Unambiguous Expression Grammar

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E)$

$F \rightarrow id$

Left Recursion

A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation.

$A \stackrel{+}{\Rightarrow} A\alpha$ for some string α

Top-down parsing techniques **cannot** handle left-recursive grammars.

So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.

The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.

Immediate Left-Recursion

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\,A'$

$A' \rightarrow \alpha\,A' \mid \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\,A' \mid \ldots \mid \beta_n\,A'$

$A' \rightarrow \alpha_1\,A' \mid \ldots \mid \alpha_m\,A' \mid \varepsilon$ an equivalent grammar

Immediate Left-Recursion -- Example

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E)$

$F \rightarrow id$

eliminate immediate left recursion $\Downarrow$

$E \rightarrow T E'$

$E' \rightarrow +T E' \mid \varepsilon$

$T \rightarrow F T'$

$T' \rightarrow *F T' \mid \varepsilon$

$F \rightarrow (E)$

$F \rightarrow id$

Algorithm 4.19: Eliminating left recursion.

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

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. $\square$

```
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)  }
```

Apply Left recursion for this example:

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

Elimination of left recursion

LHS → RHS

S → P1 | P2 | P3 | …. | Pn

Rule:

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

$$\Downarrow$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

Example 2

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

Example 2

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

$\Downarrow$

$A \rightarrow Ac \mid A\underline{ad} \mid \underline{bd} \mid \varepsilon$

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2$

$\Downarrow$

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Elimination of Left Factoring

A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

grammar ➔ a new equivalent grammar suitable for predictive parsing

stmt $\rightarrow$ if expr then stmt else stmt |

if expr then stmt

Elimination of Left-Factoring

In general,

$A \to \alpha\beta_1 \mid \alpha\beta_2$ where α is non-empty and the first symbols of β_1 and β_2 (if they have one) are different.

Elimination of Left-Factoring

But, if we re-write the grammar as follows

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Algorithm 4.21: Left factoring a grammar.

INPUT: Grammar G.

OUTPUT: An equivalent left-factored grammar.

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$\begin{aligned} A &\to \alpha A' \mid \gamma \\ A' &\to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. $\square$

Left-Factoring

Algorithm

For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

$$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$

$$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

Example 1

$S \rightarrow i\,E\,t\,S \mid i\,E\,t\,S\,e\,S \mid a$

$E \rightarrow b$

$$\Downarrow$$

$S \rightarrow i\,E\,t\,S\,S' \mid a$

$S' \rightarrow e\,S \mid \varepsilon$

$E \rightarrow b$

Example 2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$$\Downarrow$$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$$\Downarrow$$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

Example 2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

Example 3

$$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$$

$$\Downarrow$$

$$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$$

$$A' \rightarrow bB \mid B$$

$$\Downarrow$$

$$A \rightarrow aA' \mid cdA''$$

$$A' \rightarrow bB \mid B$$

$$A'' \rightarrow g \mid eB \mid fB$$

Parsing Technique

We categorize the parsers into two groups:

1. **Top-Down Parsing**
   - the parse tree is created top to bottom, starting from the root.
2. **Bottom-Up Parsing**
   - the parse is created bottom to top; starting from the leaves

Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).

Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.

- LL for top-down parsing
- LR for bottom-up parsing

Top Down Parsing

- Recursive-Descent Parsing
- Non-Recursive Predictive Parsing

Bottom Down Parsing

- Reduction
- Handle Pruning
- Shift-Reduce Parsing
- Operator Precedence
- LR Parser
  - SLR (Simple LR)
  - CLR (Canonical LR)
  - LALR (Lookahead LR)

Top Down Parsing

- Recursive-Descent Parsing
- Non-Recursive Predictive Parsing

Top Down Parser

A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right

It can be also viewed as finding a leftmost derivation for an input string

Example: id+id*id

Top-Down Parsing

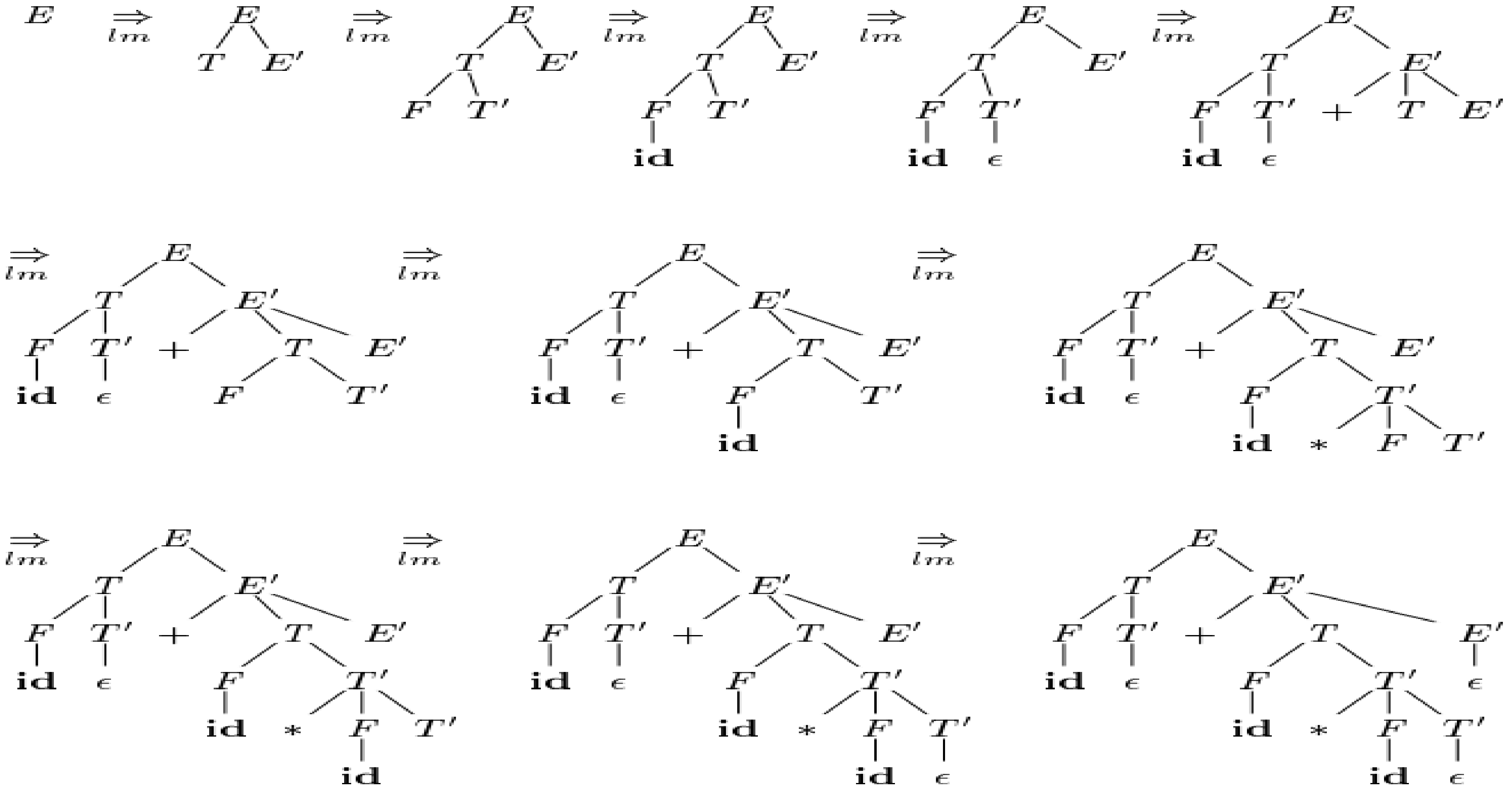


Figure 4.12: Top-down parse for **id** + **id** $*$ **id**

Recursive-Descent Parsing

void $A()$ {

Choose an A-production, $A \rightarrow X_1 X_2 \cdots X_k$;

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 */;

}

}

Recursive-Descent Parsing

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

Recursive-Descent Parsing

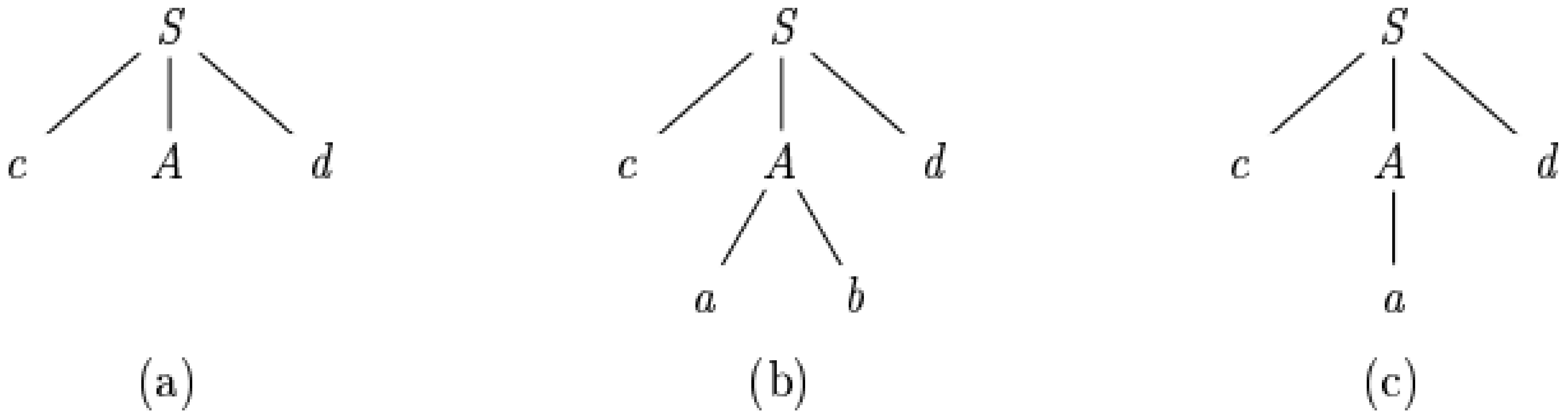


Figure 4.14: Steps in a top-down parse

Non-Recursive Predictive Parsing

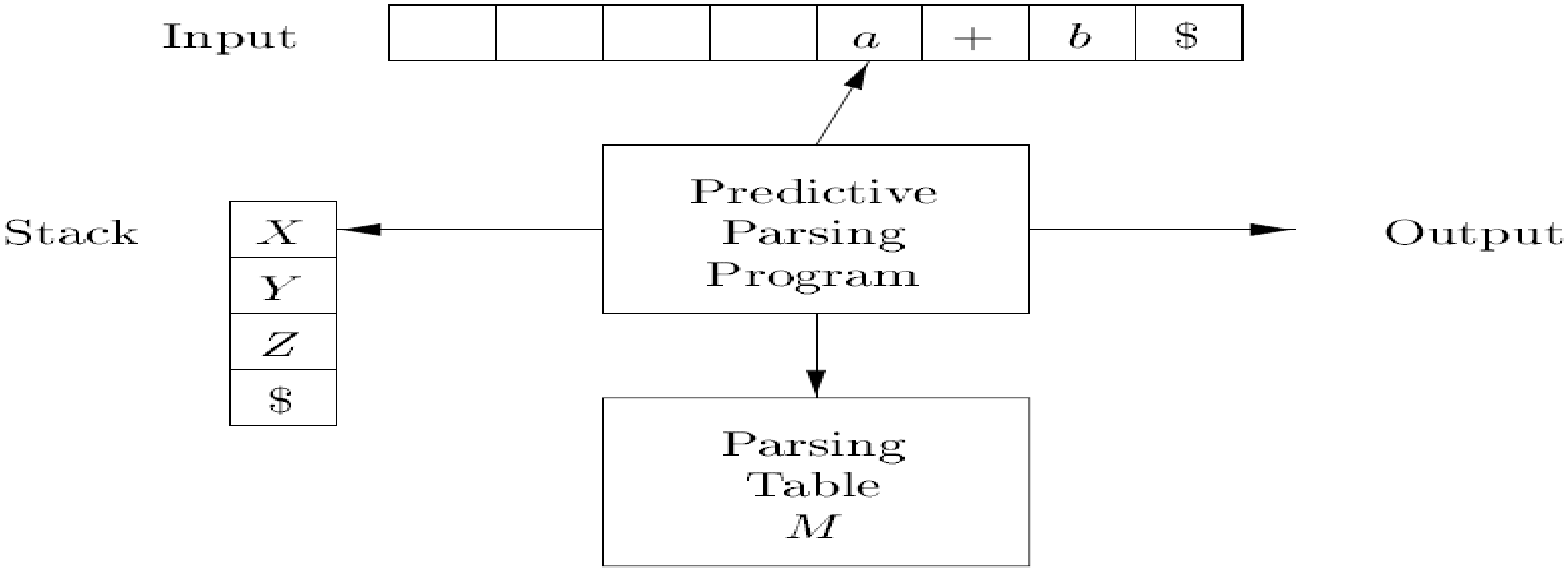


Figure 4.19: Model of a table-driven predictive parser

Non-Recursive Predictive Parsing

LL(1) Grammar

- Elimination of Left Recursion / Left Factoring
- FIRST and FOLLOW
- Predictive parsing table
- Stack Implementation

LL(1) Grammar

Predictive Parser, i.e recursive-descent parsers without backtracking is constructed for a class of grammar called LL(1)

- L – Leftmost derivation
- L – scans the input from left to right
- 1 – one input symbol of lookahead at each step to make parsing action

Elimination of Left Recursion

Unambiguous Expression Grammar :

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E)$

$F \rightarrow id$

$\Longrightarrow$

$E \rightarrow T\,E'$

$E' \rightarrow +T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *F\,T' \mid \varepsilon$

$F \rightarrow (E)$

$F \rightarrow id$

FIRST and FOLLOW

FIRST(X):

Rules:

1. X is a Terminal, FIRST (X) = { X }
2. X is a Non-terminal, $X \rightarrow Y_1, Y_2, Y_3 \ldots . Y_k$, $K \geq 1$,
   FIRST(X) = FIRST(Y_1)
3. $X \rightarrow \varepsilon$ is a production, FIRST(X) = { ε }

FIRST(x):

1) $E \rightarrow T\,E'$

FIRST (E) = FIRST (T)

2) $E' \rightarrow +T\,E' \mid \varepsilon$

$E' \rightarrow +T\,E'$

$E' \rightarrow \varepsilon$

FIRST(E') = +

FIRST (E') = ε

FIRST(x):

3) $T \rightarrow F\ T'$

FIRST(T) = FIRST(F)

4) $T' \rightarrow {*}F\ T' \mid \varepsilon$

$T' \rightarrow {*}F\ T'$

FIRST(T') = *

$T' \rightarrow \varepsilon$

FIRST(T') = ε

FIRST(x):

5) $F \rightarrow (E)$

FIRST(E) = (

6) F → id

FIRST(E) = id

FIRST (X)

Non-terminal	FIRST
E	(, id
E’	+ , ε
T	(, id
T’	* , ε
F	(, id

FIRST and FOLLOW

FOLLOW(X):

Rules:

1. S is a Start Symbol, FOLLOW(S) = $
2. If a production A → α B β , FOLLOW(B) = FIRST(β) – ε
3. If a production A → α B or A → α B β i.e, FIRST(β) = ε

   FOLLOW (B) = FOLLOW(A)

FOLLOW(X)

1) $E \rightarrow T E'$

$$\text{FOLLOW}(T) = \text{FIRST}(E') - \varepsilon$$

$$= +, \varepsilon - \varepsilon$$

$$= +$$

$$\text{FOLLOW}(T) = \text{FOLLOW}(E)$$

$$\text{FOLLOW}(T) = +, \text{FOLLOW}(E)$$

FOLLOW(X)

1) $E \rightarrow T\ E'$

FOLLOW(E') = FOLLOW(E)

FOLLOW(E') = FOLLOW(E)

FOLLOW(X)

2) $E' \rightarrow +TE' \mid \varepsilon$

FOLLOW(T) = FIRST(E')-ε

= + , ε – ε

= +

FOLLOW(T) = FOLLOW(E')

FOLLOW(T) = + , FOLLOW(E')

FOLLOW(X)

2) $E' \rightarrow +T E' \mid \varepsilon$

FOLLOW(E') = FOLLOW(E')

FOLLOW(E') = FOLLOW(E')

FOLLOW(X)

3) $T \rightarrow F\ T'$

FOLLOW(F) = FIRST(T')-ε

= * , ε – ε

= *

FOLLOW(F) = FOLLOW(T)

FOLLOW(F) = * , FOLLOW(T)

FOLLOW(X)

3) $T \rightarrow F\ T'$

FOLLOW(T') = FOLLOW(T)

FOLLOW(T') = FOLLOW(T)

FOLLOW(X)

4) $T' \rightarrow *F\ T' \mid \varepsilon$

FOLLOW(F) = FIRST(T')-ε

$= *, \varepsilon - \varepsilon$

$= *$

FOLLOW(F) = FOLLOW(T')

FOLLOW(F) = * , FOLLOW(T')

FOLLOW(X)

4) $T' \rightarrow *F\,T' \mid \varepsilon$

FOLLOW(T') = FOLLOW(T')

FOLLOW(T') = FOLLOW(T')

FOLLOW(X)

5) $F \rightarrow (E)$

FOLLOW(E) = FIRST(')') =)

FOLLOW(E) = $

FOLLOW(E) = {) , $ }

FOLLOW(X)

6) F → id

FOLLOW

Non-terminal	FOLLOW
E	) , $
E’	FOLL(E’) = FOLL(E) FOLL(E’) = FOLL(E’)
T	FOLL(T) = + , FOLL(E) FOLL(T) = +, FOLL(E’)
T’	FOLL(T’) = FOLL(T) FOLL(T’) = FOLL(T’)
F	FOLL(F) = * , FOLL(T) FOLL(F) = * , FOLL(T’)

FOLLOW

Non-terminal	**FOLLOW**	**FOLLOW**
E	) , $	) , $
E’	FOLLOW(E) , FOLLOW(E’)	) , $
T	+ , FOLLOW(E) , FOLLOW(E’)	+,) , $
T’	FOLLOW(T), FOLLOW(T’)	+,) , $
F	* , FOLLOW(T), FOLLOW(T’)	* , +,) , $

FIRST and FOLLOW

Non-terminal	FIRST	FOLLOW
E	(, id	) , $
E’	+ , ε	) , $
T	(, id	+,) , $
T’	* , ε	+,) , $
F	(, id	* , +,) , $

Non-Recursive Predictive Parsing

Algorithm 4.31: Construction of a predictive parsing table.

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(α), 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.

Predicting parsing Table

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

Figure 4.17: Parsing table M for Example 4.32

Stack Implementation

Algorithm 4.34: Table-driven 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.

Stack Implementation

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above \$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. □

```
let a be the first symbol of w;
let X be the top stack symbol;
while ( X ≠ $ ) { /* 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 → Y1 Y2 ··· Yk ) {
                output the production X → Y1 Y2 ··· Yk;
                pop the stack;
                push Yk, Yk-1, ... , Y1 onto the stack, with Y1 on top;
        }
        let X be the top stack symbol;
}
```

Table-driven Predictive Parsing

Stack	Input	Action
$E	id + id * id $	push E → TE'
$E'T	id + id * id $	push T → FT'
$E'T'F	id + id * id $	push F → id
$E'T'id	id + id * id $	pop (match id)
$E'T'	+ id * id $	push T' → ε
$E'	+ id * id $	push E' → +TE'
$E'T+	+ id * id $	pop (match +)
$E'T	id * id $	push T → FT'

Table-driven Predictive Parsing

Stack	Input	Action
$E'T'F	id * id $	push F → id
$E'T'id	id * id $	pop (match id)
$E'T'	* id$	push T' → *FT'
$E'T'F*	*id$	pop (match *)
$E'T'F	id$	push F → id
$E'T'id	id$	pop (match id)
$E'T'	$	push T' → ε
$E'	$	push E' → ε
$	$	Accept

Table-driven Predictive Parsing

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 $E' \to \epsilon$

Error Recovery in Predictive Parsing

Error is detected in predictive parsing, M[A,a] is error

- Panic mode
- Phrase-level recovery

Panic mode

- Skipping symbols on the input until a token in a selected set of synchronizing tokens appears.
- FOLLOW(A)
  - The synchronizing set for nonterminal A
  - Eg: if semicolons terminate statements as in C
- FIRST(A)
  - Add to the synchronizing set for nonterminal A
  - NT produces the empty string
  - Terminal cannot be matched, pop the terminal

Predictive Parsing Table

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

Figure 4.22: Synchronizing tokens added to the parsing table of Fig. 4.17

Parsing and Error Recovery

STACK	INPUT	REMARK
$E\ \$$	$)\ \textbf{id} * + \textbf{id}\ \$$	error, skip)
$E\ \$$	$\textbf{id} * + \textbf{id}\ \$$	**id** is in FIRST(E)
$TE'\ \$$	$\textbf{id} * + \textbf{id}\ \$$	
$FT'E'\ \$$	$\textbf{id} * + \textbf{id}\ \$$	
$\textbf{id}\ T'E'\$$	$\textbf{id} * + \textbf{id}\ \$$	
$T'E'\ \$$	$* + \textbf{id}\ \$$	
$* FT'E'\ \$$	$* + \textbf{id}\ \$$	
$FT'E'\ \$$	$+ \textbf{id}\ \$$	error, $M[F, +] =$ synch
$T'E'\ \$$	$+ \textbf{id}\ \$$	F has been popped
$E'\ \$$	$+ \textbf{id}\ \$$	
$+ TE'\ \$$	$+ \textbf{id}\ \$$	
$TE'\ \$$	$\textbf{id}\ \$$	
$FT'E'\ \$$	$\textbf{id}\ \$$	
$\textbf{id}\ T'E'\ \$$	$\textbf{id}\ \$$	
$T'E'\ \$$	$\$$	
$E'\ \$$	$\$$	
$\$$	$\$$	

Phrase-level recovery

- It is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines.
- These routines may change, insert, or delete symbols on the input and issue appropriate error messages.

Example 2

Dangling Else-Statement:

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

Input String: ibtibtaea$

Example 3

$S \to (L) \mid a$

$L \to L , S \mid S$

Input String:

(a,(a,a))$

Example 4

$S \rightarrow a \mid \uparrow \mid (T)$

$T \rightarrow T , S \mid S$

Input String:

$(a,(a,\uparrow))\$$

Thank You