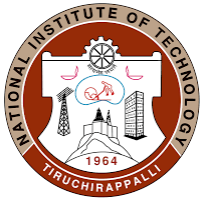


CSPC31: Principles of Programming Languages

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
Ph: 999 470 4853
E-Mail: balakrishnan@nitt.edu

Books

- **Text Books**
  - ✓ **Robert W. Sebesta, *"Concepts of Programming Languages"*, Tenth Edition, Addison Wesley, 2012.**
  - ✓ Michael L. Scott, *"Programming Language Pragmatics"*, Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
  - ✓ Allen B Tucker, and Robert E Noonan, *"Programming Languages – Principles and Paradigms"*, Second Edition, Tata McGraw Hill, 2007.
  - ✓ R. Kent Dybvig, *"The Scheme Programming Language"*, Fourth Edition, MIT Press, 2009.
  - ✓ Jeffrey D. Ullman, *"Elements of ML Programming"*, Second Edition, Prentice Hall, 1998.
  - ✓ Richard A. O'Keefe, *"The Craft of Prolog"*, MIT Press, 2009.
  - ✓ W. F. Clocksin, C. S. Mellish, *"Programming in Prolog: Using the ISO Standard"*, Fifth Edition, Springer, 2003.

Chapters

Chapter No.	Title
1.	Preliminaries
~~2.~~	~~Evolution of the Major Programming Languages~~
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages

Chapter 4 – Lexical and Syntax Analysis

Need for this Chapter

- Syntax analyzers rely on the grammars
  - Application of grammars
- Lexical and Syntax analyzers are needed in numerous situations outside compiler design
  - Compute the complexity of programs
  - Analyze and react to the contents of a configuration file

Objectives

- Introduction to Lexical Analysis
- Discusses the general parsing problem
- Recursive-descent implementation technique -> Top-down parsers
- Bottom-up Parsing -> LR Parsing Algorithm

Introduction

- Three different approaches to implement programming languages
  - Compilation, Pure Interpretation, Hybrid
- Compilation -> C++ and COBOL
- Pure Interpretation -> used for smaller systems in which execution efficiency is not critical. Eg: JavaScript
- Hybrid -> Java, Perl (JIT improves speed)
- Syntax analyzers are nearly always based on a formal description of the syntax of programs
- Most commonly used syntax-description formalism is context-free grammars or BNF

Introduction

- Reasons
  - Clear and concise
  - Can be used as the direct basis for syntax analyzer
  - Implementations are relatively easy
- Syntax Analyzing Task -> Lexical Analysis + Syntax Analysis
- Lexical Analysis -> Small-scale language constructs (Names, Numeric Literals)
- Syntax Analysis -> Large-scale constructs (Expressions, Statements and Program Units)
- Reasons for separation
  - Simplicity
  - Efficiency
  - Portability

Lexical Analysis

- Pattern matcher
- Serves as a front end of a syntax analyzer
- Collects characters into logical groupings (lexemes) and assigns internal codes to the groupings (tokens)

result = oldsum – value / 100;

Token	*Lexeme*
IDENT	result
ASSIGN_OP	=
IDENT	oldsum
SUB_OP	-
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;

Lexical Analysis

- Role of Lexical Analyzer
  - Skips comments and white space outside lexemes
  - Inserts lexemes for user-defined names into the symbol table
  - Detects syntactic errors in tokens
- Possible ways to build
  - Write a formal description of the token patterns (lex)
  - State transition diagram + Program
  - State transition diagram + Table-driven implementation

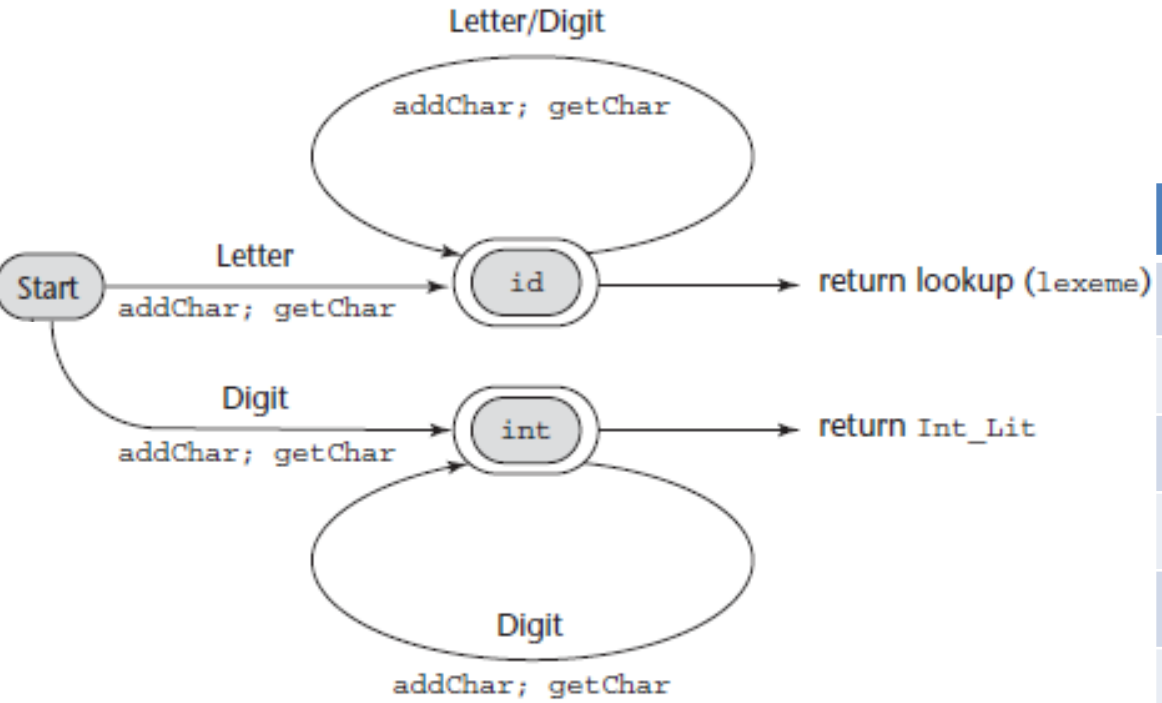
Lexical Analysis

- State Transition Diagram
  - Directed graph
  - Nodes -> State names
  - Arcs -> Labelled with input characters that cause the transitions
- State diagrams used for lexical analyzers are representations of a class of mathematical machines called finite automata
- Finite automata -> Used to recognize members of a class of languages called regular languages
- Tokens of a programming language -> Regular language
- Lexical analyzer is a finite automaton

Lexical Analysis

- Recognize program names, reserved words and int literals
- Names -> Uppercase letters, Lowercase letters and Digits
  - Must begin with a letter, No length limitation

int a;
float b;



Name	Type
a	int
b	float

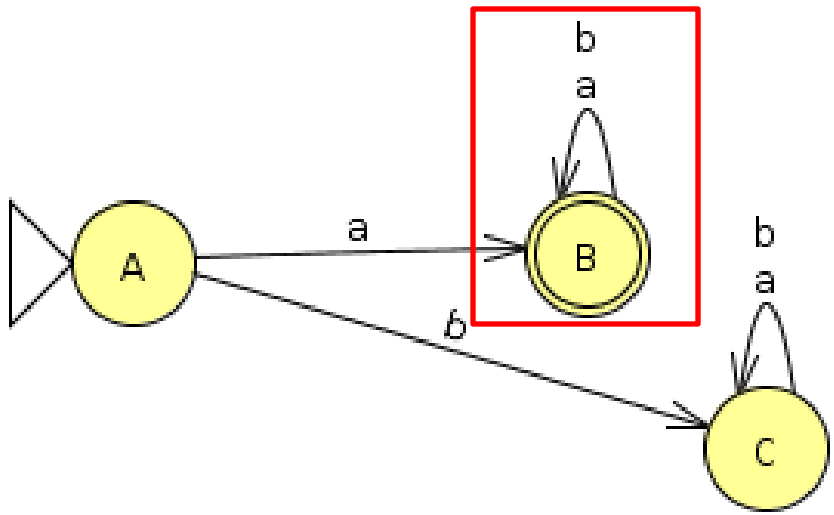
Token	Lexeme
Keyword	int
Identifier	a
Semi-colon	;
Keyword	float
Identifier	b
Semi-colon	;

Int
Float
If
Else
Char
.
.
.

Finite Automaton

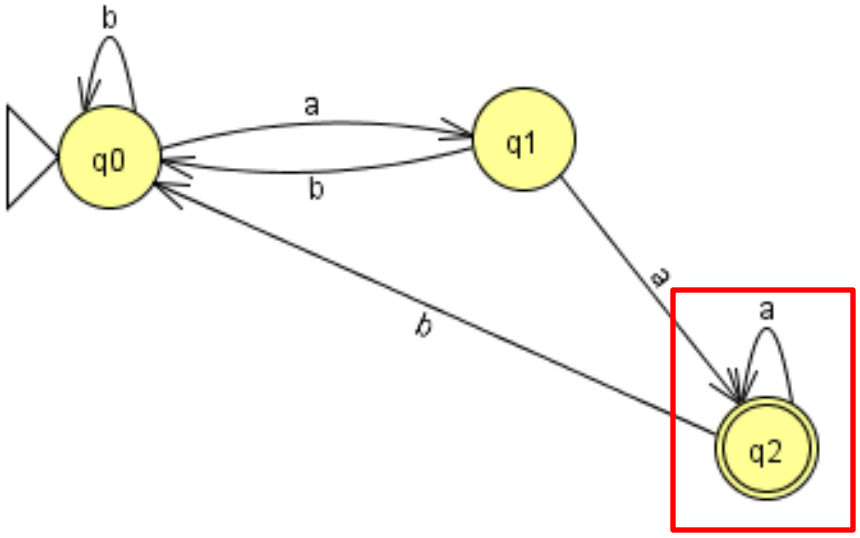
Eg 1: **a**abbbbbbabbbababababaaba
=> a.(a | b)$^+$ **(Accepted)**
Eg 2: bababababbbabaaaba
=> b.(a | b)$^+$ **(Rejected)**

This accepts all strings over *a*, *b* that start with an 'a'.



Eg 1: bbbbbbabbbbbbaaaaaaaa
=> b$^+$.(a | b)$^+$.a.a **(Accepted)**
Eg 2: bbbbbbabbbbbbaaaaaaaabbbb
=> b$^+$.(a | b)$^+$.b.b **(Rejected)**

This accepts all strings over *a*, *b* that end with two a's.



Lexical Analysis

- Responsible for the initial construction of symbol table
- User-defined names + Attributes

```
float a;
int b;
```

Variable_Name	Type
a	float
b	int

Parsing Problem / Parsing Decision Problem

- Part of the process of analyzing syntax -> Syntax Analysis or

  Parsing

- Types

  - Top-down Parsing

  - Bottom-up Parsing

Introduction to Parsing

- Goals
  - Must check the input program to determine whether it is syntactically correct
  - Produce a complete parse tree, or at least trace the structure of the complete parse tree, for syntactically correct input
- Two categories
  - Top-down
  - Bottom-up

Introduction to Parsing

- **Terminal symbols -> Lowercase letters** at the beginning of the alphabet **(a, b, . . .)**
- **Non-terminal symbols -> Uppercase letters** at the beginning of the alphabet (A, B, . . .)
- Terminals or non-terminals -> Uppercase letters at the end of the alphabet (W, X, Y, Z)
- **Strings of terminals** -> **Lowercase letters** at the end of the alphabet **(w, x, y, z)**
- **Mixed strings (terminals and/or non-terminals) -> Lowercase Greek letters (α, β, γ, δ)**

Introduction to Parsing

- For programming languages, terminal symbols are the small-scale syntactic constructs of the language, what we have referred to as lexemes

Eg: int, float, a, b

- Nonterminal symbols of programming languages are usually connotative names or abbreviations, surrounded by pointed brackets

Eg: <expr>, <A>

- Sentences of a language (programs, in the case of a programming language) are strings of terminals

Eg: if x == 5 then x = x +1 else x = x - 1, a + b

- Mixed strings describe right-hand sides (RHSs) of grammar rules and are used in parsing algorithms

Eg: if <logic_expr> then <stmt> else <stmt>

Top-Down Parsers

- Builds the parse tree in pre-order -> Leftmost Derivation
- Given a sentential form -> Finds the next sentential form in that leftmost derivation
- General notation -> xAα
  - x -> String of Terminal Symbols
  - A -> Non-terminal
  - α -> Mixed-string

if x==5 then x = x +1 else x = x-1

if <logic_expr> then <stmt> else <stmt>
x A α

logic_expr -> var == literal
| var >= literal

- Determining the next sentential form is a matter of choosing the correct grammar rule that has A as its LHS
  - A-rules: A → bB | cBb | a

Top-Down Parsers

- General notation -> xAα
  - A-rules: A → bB | cBb | a
- Different top-down parsing algorithms use different information to make parsing decisions
  - Most common top-down parsers choose the correct RHS for the leftmost nonterminal in the current sentential form by comparing the next token of input with the first symbols that can be generated by the RHSs of those rules
  - Whichever RHS has that token at the left end of the string it generates, is the correct one
- In the sentential form x**A**, the parser would use whatever token would be the first generated by A to determine which A-rule should be used to get the next sentential form
  - In the example, the three RHSs of the A-rules all begin with different terminal symbols
  - The parser can easily choose the correct RHS based on the next token of input, which must be a, b, or c in this example
- In general, choosing the correct RHS is not so straightforward, because some of the RHSs of the leftmost nonterminal in the current sentential form may begin with a nonterminal
- Types
  - Recursive-Descent Parser -> Coded version
  - Parsing Table
- **LL** Algorithm -> Left-to-Right Scan, Leftmost Derivation

Miscellaneous

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <term>
| <term>

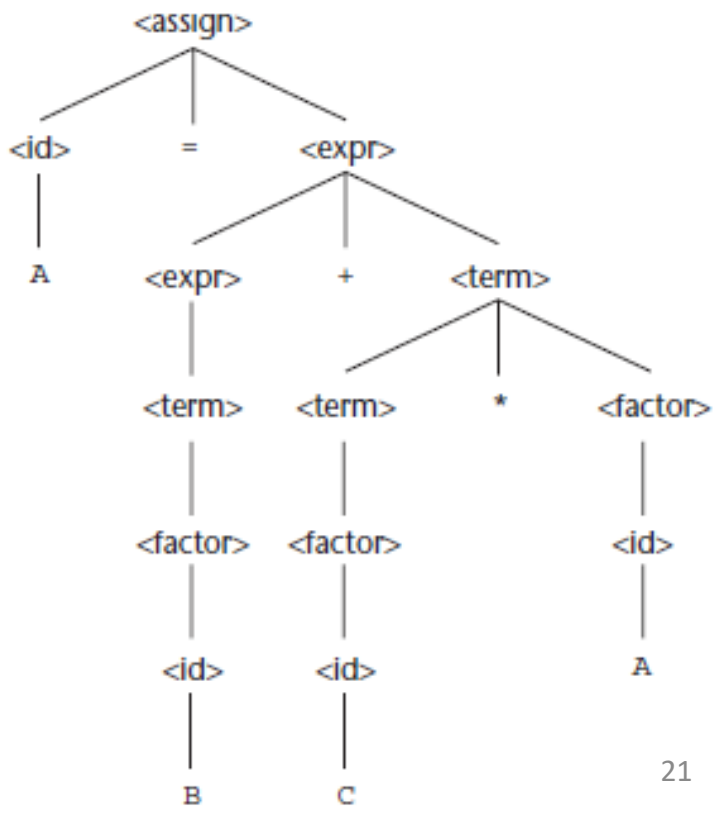
<term> → <term> * <factor>
| <factor>

<factor> → (<expr>)
| <id>

A = B + C * A

Leftmost Derivation

<assign> => <id> = <expr>
=> A = <expr>
=> A = <expr> + <term>
=> A = <term> + <term>
=> A = <factor> + <term>
=> A = <id> + <term>
=> A = B + <term>
=> A = B + <term> * <factor>
=> A = B + <factor> * <factor>
=> A = B + <id> * <factor>
=> A = B + C * <factor>
=> A = B + C * <id>
=> A = B + C * A



Bottom-Up Parsers

- Constructs a parse tree by beginning at the leaves and progressing toward the root
  - Reverse of the rightmost derivation
- Sentential forms of the derivation are produced in order of last to first
- In terms of the derivation, a bottom-up parser can be described as follows:
  - Given a right sentential form α, the parser must determine what substring of α is the RHS of the rule in the grammar that must be reduced to its LHS to produce the previous sentential form in the rightmost derivation
  - Eg: First step for a bottom-up parser is to determine which substring of the initial given sentence is the RHS to be reduced to its corresponding LHS to get the second last sentential form in the derivation

Input: aabc

$S \rightarrow aAc$
$A \rightarrow aA \mid b$

$S \Rightarrow aAc \Rightarrow aaAc \Rightarrow aabc$

S → aabc
S → aaAc ←
S → aAc

Bottom-Up Parsers

- Process of finding the correct RHS to reduce is complicated by the fact that a given right sentential form may include more than one RHS from the grammar of the language being parsed
- Correct RHS is called the **handle**

Top-Down Parser
Recursive-Descent Parsing

- Consists of a collection of subprograms, many of which are recursive, and it produces a parse tree in top-down order

- EBNF is ideally suited for recursive-descent parsers

  - {} -> 0 or more times

  - [] -> once or not at all

<if_statement> → **if** <logic_expr> <statement> [**else** <statement>]
<ident_list> → ident {, ident}

Recursive-Descent Parsing

- A recursive-descent parser has a subprogram for each non-terminal in its associated grammar
- When given an input string, it traces out the parse tree that can be rooted at that non-terminal and whose leaves match the input string
- A recursive-descent parsing subprogram is a parser for the language (set of strings) that is generated by its associated non-terminal

<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int_constant | (<expr>)

Recursive Descent Parsing

<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int_constant | (<expr>)

```
/* expr
   Parses strings in the language generated by the rule:
   <expr> -> <term> {(+ | -) <term>}
   */
void expr() {
  printf("Enter <expr>\n");

/* Parse the first term */
  term();

/* As long as the next token is + or -, get
   the next token and parse the next term */
  while (nextToken == ADD_OP || nextToken == SUB_OP) {
    lex();
    term();
  }
  printf("Exit <expr>\n");
}  /* End of function expr */
```

a + b

Does not include code for syntax error detection or recovery

Recursive Descent Parsing

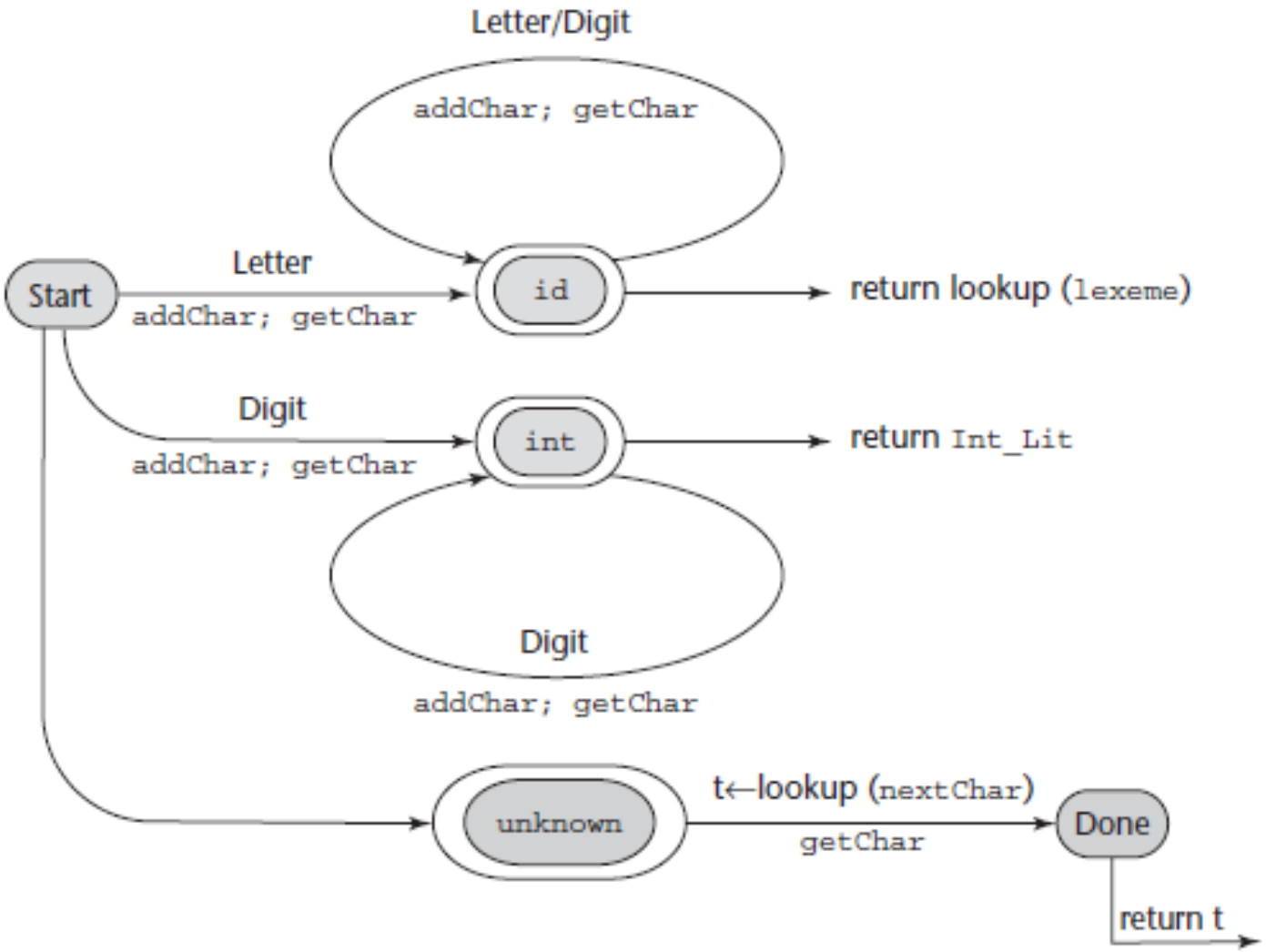
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int_constant | (<expr>)

```
/* term
   Parses strings in the language generated by the rule:
   <term> -> <factor> {(* | /) <factor>)
   */
void term() {
  printf("Enter <term>\n");

/* Parse the first factor */
  factor();

/* As long as the next token is * or /, get the
   next token and parse the next factor */
  while (nextToken == MULT_OP || nextToken == DIV_OP) {
    lex();
    factor();
  }
  printf("Exit <term>\n");
}  /* End of function term */
```

Recursive Descent Parsing

```
/* factor
   Parses strings in the language generated by the rule:
   <factor> -> id | int_constant | ( <expr )
   */
void factor() {
  printf("Enter <factor>\n");

/* Determine which RHS */
  if (nextToken == IDENT || nextToken == INT_LIT)

/* Get the next token */
    lex();

/* If the RHS is ( <expr>), call lex to pass over the
   left parenthesis, call expr, and check for the right
   parenthesis */
  else {
    if (nextToken == LEFT_PAREN) {
      lex();
      expr();
      if (nextToken == RIGHT_PAREN)
        lex();
      else
        error();
    }  /* End of if (nextToken == ... */

/* It was not an id, an integer literal, or a left
   parenthesis */
    else
      error();
  }  /* End of else */
```

Recursive Descent Parsing

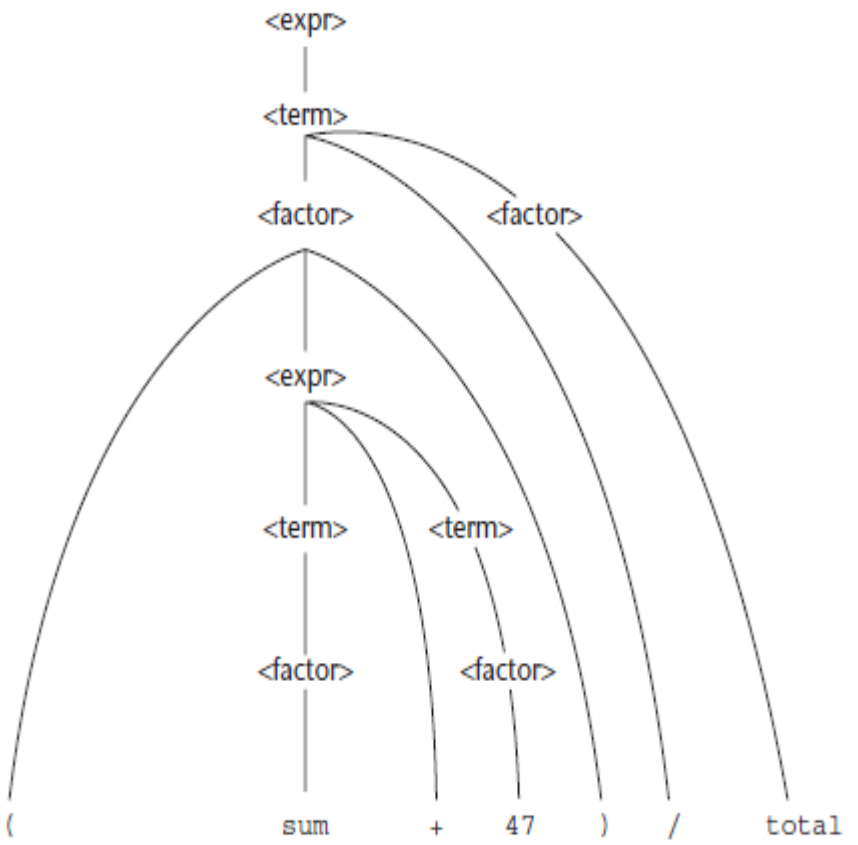


Recursive Descent Parsing

(sum + 47) / total

```
Next token is: 25 Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11 Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21 Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26 Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24 Next lexeme is /
Exit <factor>
Next token is: 11 Next lexeme is total
Enter <factor>
Next token is: -1 Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int_constant | (<expr>)



LL Grammar Class

- Left Recursion is a problem for LL Parsers (Recursive Descent)

$$A \rightarrow A + B$$

- A recursive-descent parser subprogram for A immediately calls itself to parse the first symbol in its RHS
- That activation of the A parser subprogram then immediately calls itself again, and again, and so forth
- **Two Types:** Direct and Indirect Left Recursion
- Direct Left Recursion

ε specifies the empty string. A rule that has ε as its RHS is called an erasure rule, because its use in a derivation effectively erases its LHS from the sentential form.

For each nonterminal, A,

1. Group the A-rules as $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ where none of the β's begins with A
2. Replace the original A-rules with

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_m A' \mid \varepsilon$$

ε → Erasure Rule

LL Grammar Class

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

For the E-rules, we have $\alpha_1 = + T$ and $\beta = T$, so we replace the E-rules with

$E \rightarrow T\ E'$

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

For the T-rules, we have $\alpha_1 = * F$ and $\beta = F$, so we replace the T-rules with

$T \rightarrow F\ T'$

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

LL Grammar Class

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

=>

$$E \rightarrow T\,E'$$
$$E' \rightarrow +\,T\,E' \mid \varepsilon$$
$$T \rightarrow F\,T'$$
$$T' \rightarrow *\,F\,T' \mid \varepsilon$$
$$F \rightarrow (E) \mid id$$

LL Grammar Class

- Indirect Left Recursion also poses same problem

$$A \rightarrow BaA$$

$$B \rightarrow Ab$$

- A recursive-descent parser for these rules would have the A subprogram immediately call the subprogram for B, which immediately calls the A subprogram
  - Indirect Left Recursion -> Solution exists (not covered here)
- Problem for all top-down parsing algorithms
- Left recursion is not a problem for bottom-up parsing algorithms

LL Grammar Class

- Left recursion is not the only grammar trait that disallows top-down parsing
- Another is whether the parser can always choose the correct RHS on the basis of the next token of input, using only the first token generated by the leftmost nonterminal in the current sentential form
- There is a relatively simple test of a non–left recursive grammar that indicates whether this can be done, called the pairwise disjointness test

<expr> → **<expr> + <term>**

| **<term>**

- Pairwise Disjointness Set -> Which RHS to choose

$$\text{FIRST}(\alpha) = \{a \mid \alpha =>^* a\beta\} \text{ (If } \alpha =>^* \varepsilon, \varepsilon \text{ is in FIRST}(\alpha))$$

in which $=>^*$ means 0 or more derivation steps.

LL Grammar Class

The pairwise disjointness test is as follows:

For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that

$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$$

(The intersection of the two sets, $FIRST(\alpha_i)$ and $FIRST(\alpha_j)$, must be empty.)

- If a non-terminal A has more than one RHS, the first terminal symbol that can be generated in a derivation for each of them must be unique to that RHS

A → aB | bAb | Bb

B → cB | d

- FIRST sets for the RHSs of the A-rules are {a}, {b}, {c, d}

LL Grammar Class

A → aB | BAb

B → aB | b

- FIRST sets for the RHSs in the A-rules are {a}, {a, b} -> Fails the test
- In many cases, a grammar that fails the pairwise disjointness test can be modified so that it will pass the test

<variable> → identifier | identifier [<expression>]

- Do not pass pairwise disjointness test
- This problem can be alleviated through a process called left factoring
- Parts that follow identifier in the two RHSs are (the empty string) and [<expression>]

<variable> → identifier <new>

<new> → ε | [<expression>]

- EBNF: <variable> → identifier [[<expression]]
- Left factoring cannot solve all pairwise disjointness problems of grammars
- Sometimes, rules must be rewritten in other ways to eliminate the problem

Bottom-Up Parsing - Parsing Problem

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

id + id * id

- **LL** Algorithm -> Left-to-Right Scan, Rightmost Derivation in Reverse
- Left-Recursion -> No problem
- No metasymbols

Rightmost Derivation in Reverse

E => E + T
E + T * F
E + T * id
E + F * id
E + id * id
T + id * id
F + id * id
id + id * id

Bottom-Up Parsing - Parsing Problem

- Underlined part of each sentential form is the RHS that is rewritten as its corresponding LHS to get the previous sentential form
- Bottom-up parsing produces the reverse of a rightmost derivation
- Task is to find the specific RHS that must be rewritten to get the next (previous) sentential form

E + T * id

- Three RHSs, E + T, T, id -> Only one of these is the handle
- If E + T is chosen => E * id -> Not a legal right sentential form
- Handle of a right sentential form is unique
- Task of a bottom-up parser is to find the handle of any given right sentential form that can be generated by its associated grammar

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

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

Bottom-Up Parsing - Parsing Problem

$=>^*_{rm}$ specifies zero or more rightmost derivation steps

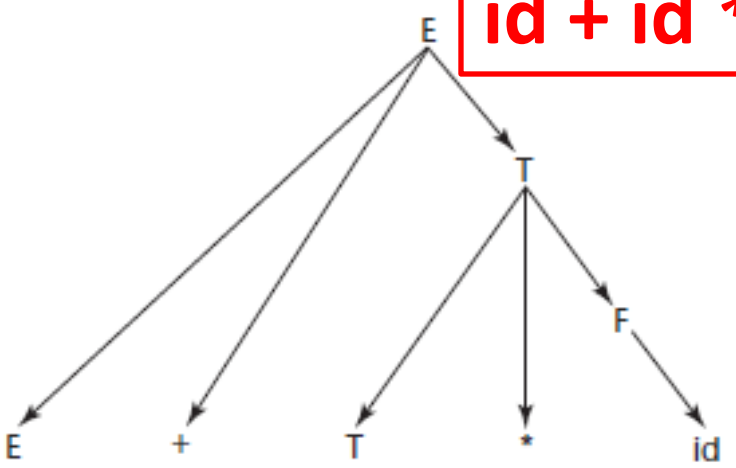
Definition: β is the **handle** of the right sentential form $\gamma = \alpha\beta w$ if and only if $S =>^*_{rm} \alpha A w =>_{rm} \alpha\beta w$

Definition: β is a **phrase** of the right sentential form γ if and only if $S =>^* \gamma = \alpha_1 A \alpha_2 =>^+ \alpha_1 \beta \alpha_2$

$=>^+_{rm}$ specifies one or more rightmost derivation steps

Definition: β is a **simple phrase** of the right sentential form γ if and only if $S =>^* \gamma = \alpha_1 A \alpha_2 => \alpha_1 \beta \alpha_2$

id + id * id



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

Phrase:

- E + T * id
- T * id
- id

Simple Phrase:

- id

Introduction to Parsing

- **Terminal symbols -> Lowercase letters** at the beginning of the alphabet **(a, b, . . .)**
- **Non-terminal symbols -> Uppercase letters** at the beginning of the alphabet (A, B, . . .)
- Terminals or non-terminals -> Uppercase letters at the end of the alphabet (W, X, Y, Z)
- **Strings of terminals** -> **Lowercase letters** at the end of the alphabet **(w, x, y, z)**
- **Mixed strings (terminals and/or non-terminals) -> Lowercase Greek letters (α, β, γ, δ)**

Bottom-Up Parsing - Parsing Problem

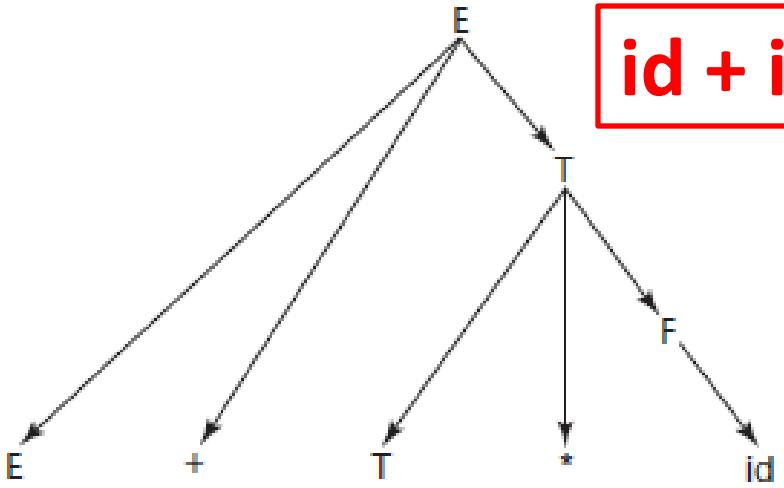
The definitions of phrase and simple phrase may appear to have the same lack of practical value as that of a handle, but that is not true. Consider what a phrase is relative to a parse tree. It is the string of all of the leaves of the partial parse tree that is rooted at one particular internal node of the whole parse tree. A simple phrase is just a phrase that takes a single derivation step from its root nonterminal node. In terms of a parse tree, a phrase can be derived from a single nonterminal in one or more tree levels, but a simple phrase can be derived in just a single tree level. Consider the parse tree shown in Figure 4.3.

Level 0

Level 1

Level 2

Level 3



id + id * id

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

Phrase:

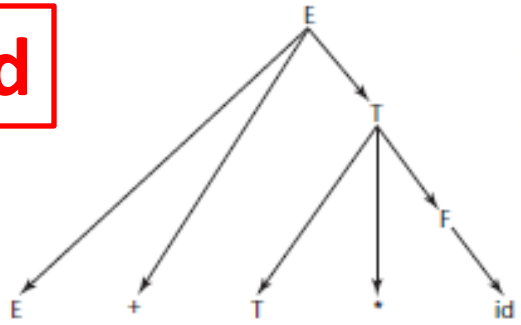
- E + T * id
- T * id
- id

Simple Phrase:

- id

Bottom-Up Parsing - Parsing Problem

id + id * id



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

Phrase:

- E + T * id
- T * id
- id

Simple Phrase:

- id

The leaves of the parse tree in Figure 4.3 comprise the sentential form E + T * id. Because there are three internal nodes, there are three phrases. Each internal node is the root of a subtree, whose leaves are a phrase. The root node of the whole parse tree, E, generates all of the resulting sentential form, E + T * id, which is a phrase. The internal node, T, generates the leaves T * id, which is another phrase. Finally, the internal node, F, generates id, which is also a phrase. So, the phrases of the sentential form E + T * id are E + T * id, T * id, and id. Notice that phrases are not necessarily RHSs in the underlying grammar.

The simple phrases are a subset of the phrases. In the previous example, the only simple phrase is id. A simple phrase is always an RHS in the grammar.

The reason for discussing phrases and simple phrases is this: The handle of any rightmost sentential form is its leftmost simple phrase. So now we have a highly intuitive way to find the handle of any right sentential form, assuming we have the grammar and can draw a parse tree. This approach to finding handles is of course not practical for a parser. (If you already have a parse tree, why do you need a parser?) Its only purpose is to provide the reader with some intuitive feel for what a handle is, relative to a parse tree, which is easier than trying to think about handles in terms of sentential forms.

We can now consider bottom-up parsing in terms of parse trees, although the purpose of a parser is to produce a parse tree. Given the parse tree for an entire sentence, you easily can find the handle, which is the first thing to rewrite in the sentence to get the previous sentential form. Then the handle can be pruned from the parse tree and the process repeated. Continuing to the root of the parse tree, the entire rightmost derivation can be constructed.

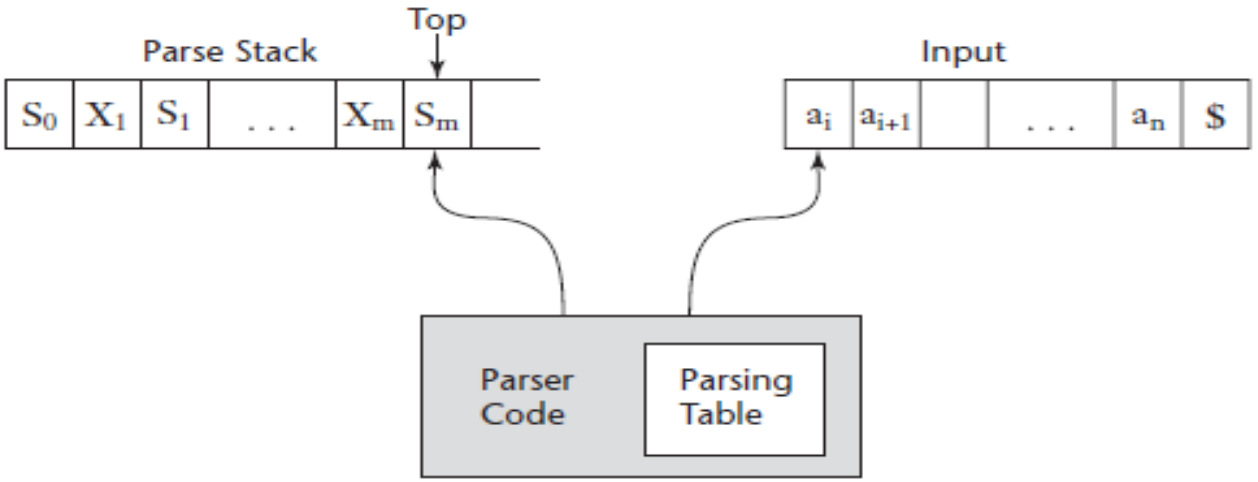
Bottom-Up Parsing – Parsing Problem

- Phrases are not necessarily RHSs in the underlying grammar
- Simple phrase is always an RHS in the grammar
- Handle of any rightmost sentential form is its leftmost simple phrase
  - Replace id with F
- Consider bottom-up parsing in terms of parse trees
  - Given the parse tree for an entire sentence, you easily can find the handle
  - Rewrite in the sentence and get the previous sentential form
  - Handle can be pruned from the parse tree and the process repeated
- Continue to the root of the parse tree

Shift-Reduce Algorithms

- Bottom-up parsers are often called shift-reduce algorithms
- Two Operations: Shift and Reduce
- Shift -> Moves the next input token onto the parser's stack
- Reduce -> Replaces an RHS (the handle) on top of the parser's stack by its corresponding LHS
- Every parser for a programming language is a pushdown automaton (PDA), because a PDA is a recognizer for a context-free language

PDA



	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

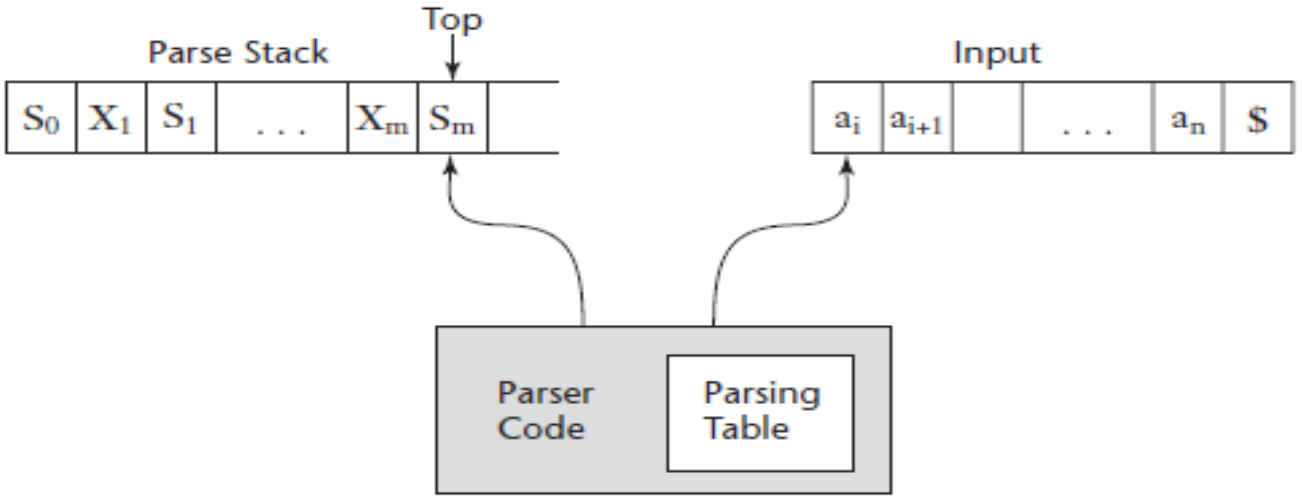
Shift-Reduce Algorithms - PDA

- PDA is a very simple mathematical machine that scans strings of symbols from left to right
- Uses a pushdown stack as its memory
- PDAs can be used as recognizers for context-free languages
- Given a string of symbols over the alphabet of a context-free language, a PDA that is designed for the purpose can determine whether the string is or is not a sentence in the language
- In the process, the PDA can produce the information needed to construct a parse tree for the sentence

LR Parsers

- Many different bottom-up parsing algorithms have been devised
- Most of them are variations of a process called LR
- LR parsers use a relatively small program and a parsing table that is built for a specific programming language
- Advantages of LR Parsers
  - Can be built for all programming languages
  - Can detect syntax errors as soon as it performs a left-to-right scan
  - LR class of grammars is a proper superset of the class parsable by LL parsers
- Disadv: Producing the Parsing Table by hand

LR Parsers



- Contents of the parse stack for an LR parser have the following form: $S_0X_1S_1X_2…X_mS_m$ (Top)
  - S -> State Symbols; X -> Grammar Symbols
  - An LR parser configuration is a pair of strings (Stack, Input):

$$(S_0X_1S_1X_2S_2…X_mS_m, a_ia_{i+1}…a_n\$)$$

LR Parsers

- Two Parts: ACTION, GOTO
- Action Part
  - State symbols -> Row labels; Terminal symbols -> column labels
- Current parse state + Next input symbol
- Actions: Shift, Reduce, Accept, Error
  - Shift -> Shifts the next input symbol onto the parse stack
  - Reduce -> Handle on top of the stack, which it reduces to the LHS of the rule whose RHS is the same as the handle

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

LR Parsers

- Goto Part
  - State symbols as labels
  - Has non-terminals as column labels
  - Values in the GOTO part indicate which state symbol should be pushed onto the parse stack after a reduction has been completed
  - The specific symbol is found at the row whose label is the state symbol on top of the parse stack after the handle and its associated state symbols have been removed
  - Column of the GOTO table that is used is the one with the label that is the LHS of the rule used in the reduction

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

$E \Rightarrow \underline{E + T}$
$\Rightarrow E + \underline{T * F}$
$\Rightarrow E + T * \underline{id}$
$\Rightarrow E + \underline{F} * id$
$\Rightarrow E + \underline{id} * id$
$\Rightarrow \underline{T} + id * id$
$\Rightarrow \underline{F} + id * id$
$\Rightarrow \underline{id} + id * id$

Example

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

$E \Rightarrow \underline{E + T}$
$\Rightarrow E + \underline{T * F}$
$\Rightarrow E + T * \underline{id}$
$\Rightarrow E + \underline{F} * id$
$\Rightarrow E + \underline{id} * id$
$\Rightarrow \underline{T} + id * id$
$\Rightarrow \underline{F} + id * id$
$\Rightarrow \underline{id} + id * id$

Stack	*Input*	*Action*
0	id + id * id $	Shift 5
0id5	+ id * id $	Reduce 6 (use GOTO[0, F])
0F3	+ id * id $	Reduce 4 (use GOTO[0, T])
0T2	+ id * id $	Reduce 2 (use GOTO[0, E])
0E1	+ id * id $	Shift 6
0E1+6	id * id $	Shift 5
0E1+6id5	* id $	Reduce 6 (use GOTO[6, F])
0E1+6F3	* id $	Reduce 4 (use GOTO[6, T])
0E1+6T9	* id $	Shift 7
0E1+6T9*7	id $	Shift 5
0E1+6T9*7id5	$	Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10	$	Reduce 3 (use GOTO[6, T])
0E1+6T9	$	Reduce 1 (use GOTO[0, E])
0E1	$	Accept

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Miscellaneous

Stack	*Input*	*Action*
0	id + id * id $	Shift 5
0id5	+ id * id $	Reduce 6 (use GOTO[0, F])
0F3	+ id * id $	Reduce 4 (use GOTO[0, T])
0T2	+ id * id $	Reduce 2 (use GOTO[0, E])
0E1	+ id * id $	Shift 6
0E1+6	id * id $	Shift 5
0E1+6id5	* id $	Reduce 6 (use GOTO[6, F])
0E1+6F3	* id $	Reduce 4 (use GOTO[6, T])
0E1+6T9	* id $	Shift 7
0E1+6T9*7	id $	Shift 5
0E1+6T9*7id5	$	Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10	$	Reduce 3 (use GOTO[6, T])
0E1+6T9	$	Reduce 1 (use GOTO[0, E])
0E1	$	Accept