BHCS15B: System Programming

Syntax Analysis

Mahesh Kumar
(maheshkumar@andc.du.ac.in)

Course Web Page
(www.mkbhandari.com/mkwiki)

Outline

1. Role of a Syntax Analyzer (or Parser)
2. Context Free Grammars
3. Derivation and Parse Tree (online tutorial)
4. Bottom-up Parsing
5. LR Parsing (Handwritten notes + online tutorial)
6. YACC (Handwritten notes + online tutorial)

Compilation Phases (revisited)

- Compilation phases are divided into several phases:
  1. Lexical Analysis (Scanning)
  2. Syntax Analysis (Parsing)
  3. Semantic Analysis
  4. Intermediate Code Generation
  5. Code Optimization
  6. Code Generation

Compilation Phases (revisited)

- Compilation phases are divided into several phases:

1. Lexical Analysis (Scanning)
2. Syntax Analysis (Parsing)
3. Semantic Analysis
4. Intermediate Code Generation

Front-end pass (One Pass)

5. Code Optimization

An Optional Pass

6. Code Generation

A Back-end pass (another Pass)

Role of a Parser

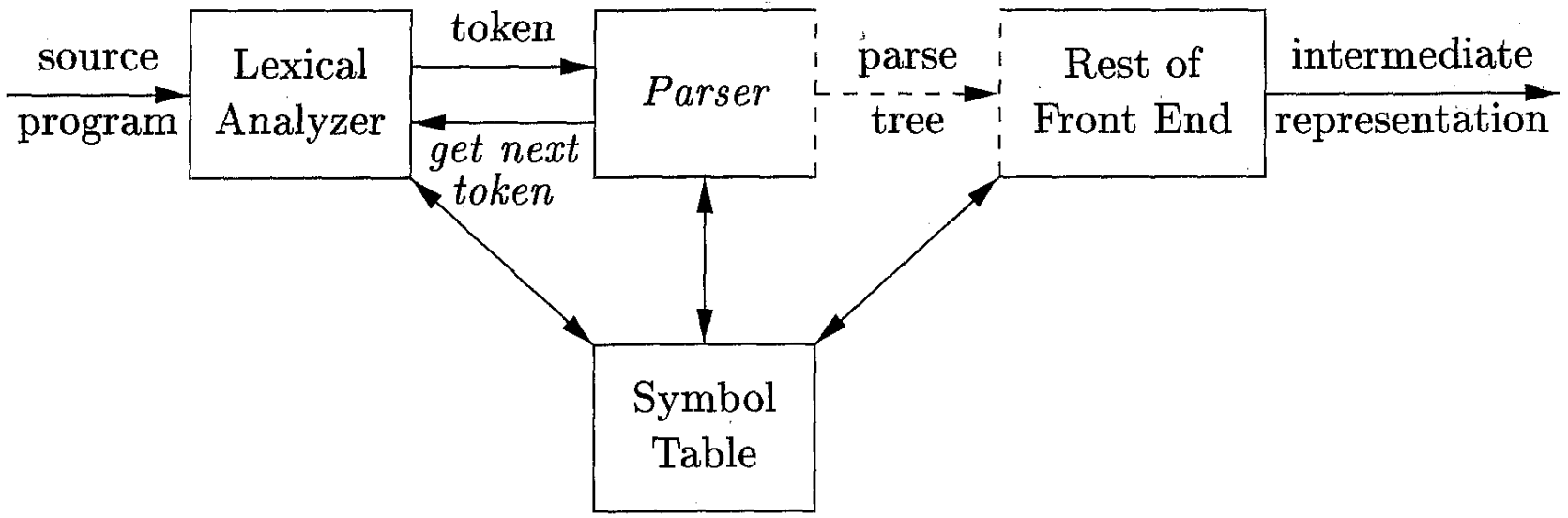
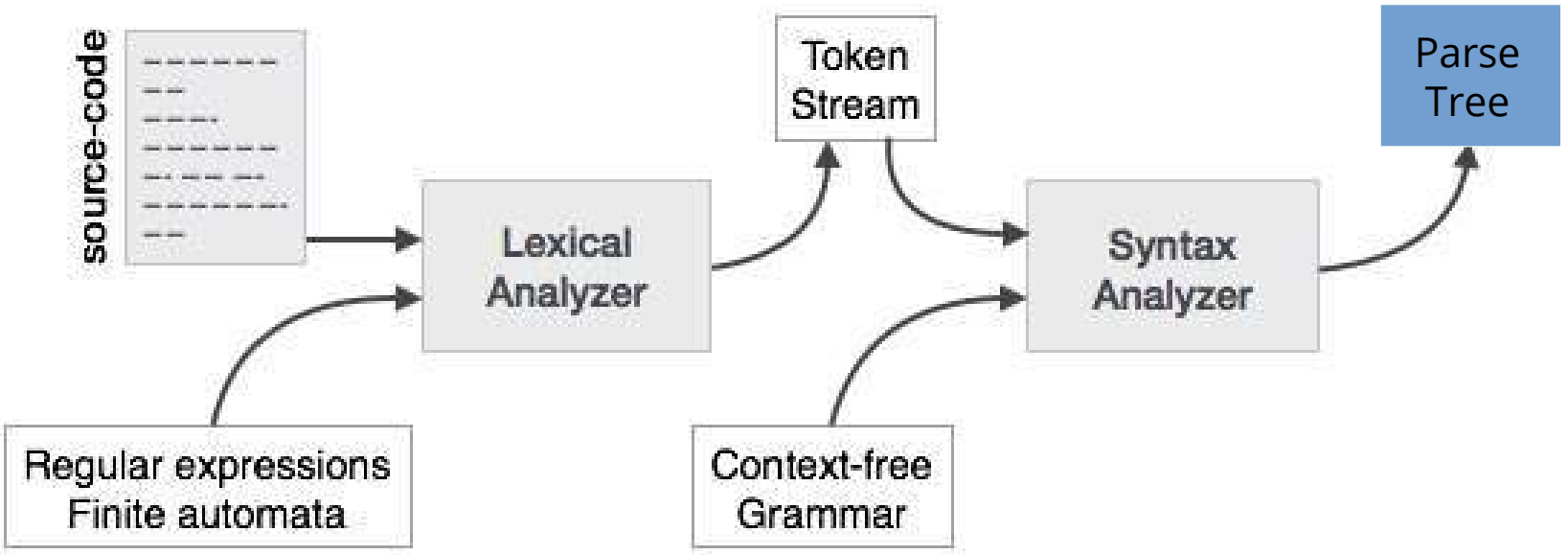


Figure 4.1: Position of parser in compiler model

Role of a Parser



Types of Parsers

- The methods commonly used in compilers are:

1. Universal
   - → *Can parse any grammar*
   - → *Too inefficient to use in production compilers*
2. Top-down
   - → *Parse-trees built from root to leaves*
   - → *Input to parser scanned from left to right one symbol at a time*
3. Bottom-up
   - → *Start from leaves and work their way up to the root*
   - → *Input to parser scanned from left to right one symbol at a time*

Syntax Error Handling

- Common programming errors can occur at many different levels.

1. Lexical Errors
   → *include misspellings of identifiers, keywords, or operators.*

2. Syntactic errors
   → *Include misplaced semicolons or extra or missing braces;*

3. Semantic errors
   → *incompatible value assignment or type mismatches between operator and operand*

4. Logical errors
   → *code not reachable, infinite loop.*

Syntax Error Handling (2)

- Identify the type of errors in the following example:

a. 12ab

b. (a+(b*c)

c. 2 + a[i]

d.
```
for(; ;) {
        printf("Hello \n");
}
```

Syntax Error Handling (2)

- Identify the type of errors in the following example:

a. 12ab // lexical error

b. (a+(b*c) // syntactic error

c. 2 + a[i] // semantic error

d.
```
for(; ;) {
        printf("Hello \n");          // logical error
}
```

Syntax Error Handling (3)

- The error handler in a parser has goals that are simple to state but challenging to realize:
  - Report the presence of errors clearly and accurately.
  - Recover from each error quickly enough to detect subsequent errors.
  - Add minimal overhead to the processing of correct programs.

Error-Recovery Strategies

- Common programming errors can occur at many different levels.

1. Panic mode:
   - → *the parser discards input symbols one at a time until one of a designated set of synchronizing tokens (delimiters like ',' , ';') is found.*
2. Phrase level
   - → *Parser may perform local correction on the remaining input. For example: replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.*
3. Error productions:
   - → *we augment the error productions to construct a parser. Error diagnostics can be generated to indicate the erroneous construct.*
4. Global correction
   - → *a minimal sequence of changes to obtain a globally least-cost correction.*

Context Free Grammar

- A Context Free Grammar is defined in the form of quadruple:

$$G = (V, T, P, S)$$

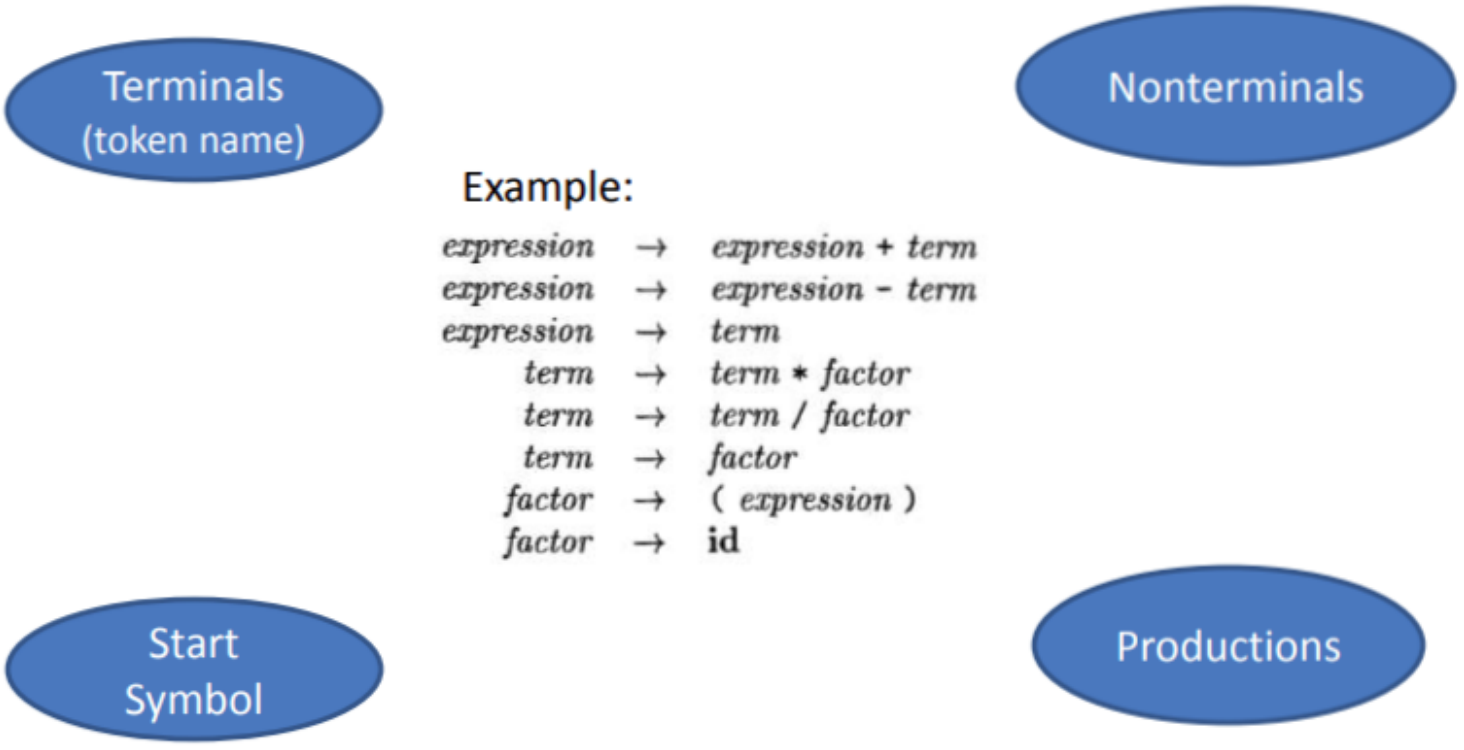
  - **V** is a finite set of non-terminals(syntactic variables)
  - **T** is a finite set of terminals
  - **P** is finite set of production rules in the form $\mathbf{A \rightarrow \alpha}$
    - → ***A*** *is a non-terminal symbol*
    - → $\mathbf{\alpha \in (V \cup T)^*}$ (any occurrences of non-terminals and terminals including 0 occurrence, i.e., epsilon)
  - **S** is a start symbol (a non-terminal symbol)

Capabilities of CFG

- There are the various capabilities of CFG:
  - Context free grammar is useful to describe most of the programming languages.
  - If the grammar is properly designed then an efficient parser can be constructed automatically.
  - Using the features of associativity & precedence information, suitable grammars for expressions can be constructed.
  - Context free grammar is capable of describing nested structures like: balanced parentheses, matching begin-end, corresponding if-then-else's & so on.

Context Free Grammar (2)

- Grammar for simple arithmetic expressions



CFG (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.

CFG (Notational Conventions)

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.

CFG (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 $A \to \alpha$, where A is the head and α the body.

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

CFG (Notational Conventions)

- From the given grammar, identify terminals, nonterminals, and start symbol.

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

CFG (Notational Conventions)

- From the given grammar, identify terminals, nonterminals, and start symbol.

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

  - Nonterminals: **E, T, F**
  - Start Symbol: **E**
  - Terminals: **id, *, + , -, /**

Bottom-Up Parsing

- The construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

> *Bottom-up parsing is the process of reducing input string to the starting symbol of the grammar*

- It generates right-most derivation in reverse order.
- A general style of bottom-up parsing known as shift-reduce parsing.
- LR grammars are the largest class of grammars for which shift-reduce parsers can be built.

Bottom-Up Parsing (2)

- The sequence of tree snapshots (*in next slide*) illustrates a bottom-up parse of the token stream **id * id**, with respect to the expression grammar (G1)

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

E represents expressions consisting of terms separated by **+** signs,

T represents terms consisting of factors separated by ***** signs, and

F represents factors that can be either parenthesized expressions or identifiers

Bottom-Up Parsing (3)

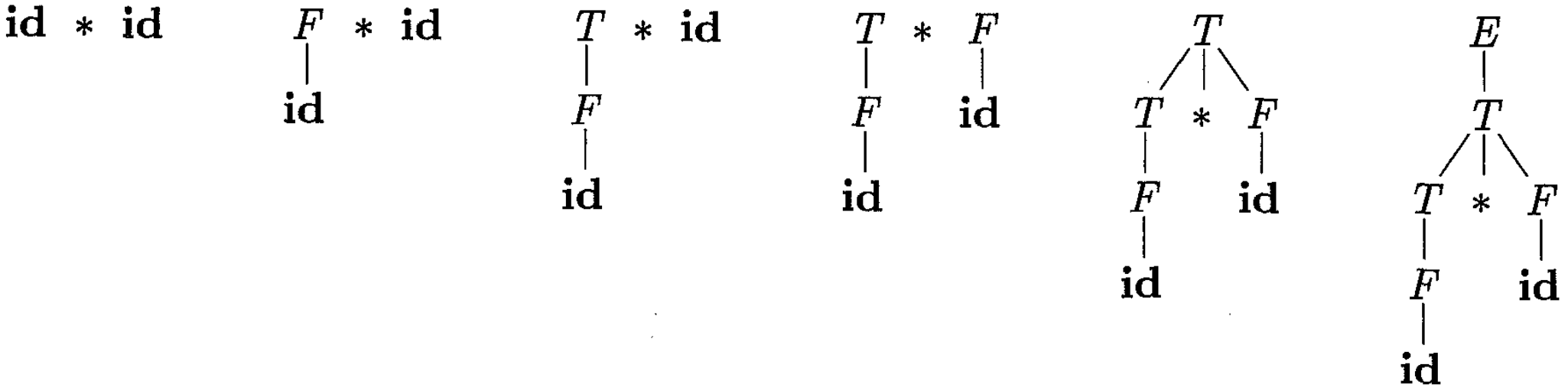


Figure 4.25: A bottom-up parse for **id * id**

Reductions

- We can think of Bottom-up parsing as the process of "reducing" a string ***w*** to the start symbol of the grammar.

- At each *reduction step*, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

- The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

Reduction means if the substring (or handle) matches with right hand side of the production then it is reduced to the corresponding left hand side non-terminal.

Reductions (2)

- *The following* illustrates a sequence of reductions (*in terms of the sequence of strings*)

$$\mathbf{id} * \mathbf{id}, \quad F * \mathbf{id}, \quad T * \mathbf{id}, \quad T * F, \quad T, \quad E$$

- By definition, a reduction is the reverse of a step in a derivation
- The goal of bottom-up parsing is therefore to construct a derivation in reverse.
- The following derivation corresponds to the parse in *Figure 4.25*

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

Reductions (2)

- *The following* illustrates a sequence of reductions (*in terms of the sequence of strings*)

$$\mathbf{id} * \mathbf{id}, \quad F * \mathbf{id}, \quad T * \mathbf{id}, \quad T * F, \quad T, \quad E$$

- By definition, a reduction is the reverse of a step in a derivation
- The goal of bottom-up parsing is therefore to construct a derivation in reverse.
- The following derivation corresponds to the parse in *Figure 4.25*

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

(a rightmost derivation)

Handle and Handle Pruning

- Bottom-up parsing during a left-to-right scan of the input constructs a right-most derivation in reverse.

> *Handle is a substring which matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.*

- The handle during parsing for input string **id * id**, is shown in the next slide. Consider the following Grammar (production rules):

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

Handle and Handle Pruning (2)

A rightmost derivation in reverse can be obtained by "handle pruning".

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

Figure 4.26: Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

Class Assignment

Q1 For the grammar shown below, and input string "abbcde", indicate the handle in right-sentential forms:

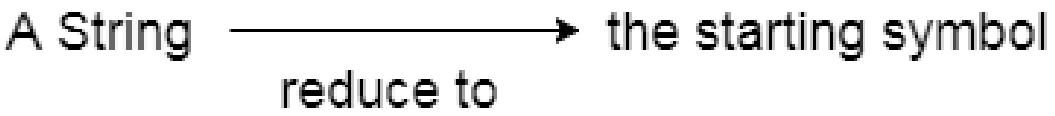
S → aABe
A → Abc | b
B → d

Q2 *For the grammar* ***S → S S + | S S * | a*** indicate the handle in each of the following right-sentential forms:

a) *S S S + a * +*
b) *S S + a * a +*
c) *a a a * a + +*

Shift-Reduce Parsing

- Shift-reduce parsing is a form of bottom-up parsing.



- It uses a *stack* to hold the grammar and an *input buffer* to hold the string (*rest of the string to be parsed*).

- There are actually four possible actions a shift-reduce parser can make:
  1. *Shift*
  2. *Reduce*
  3. *Accept*
  4. *Error*

Shift-Reduce Parsing (2)

1. *Shift* - A push operaton, shifts the next input symbol onto the top of the stack.

2. *Reduce* – If top of the stack (substring / handle) matches with the right side of the production then it is reduced to corresponding left side nonterminal.

3. *Accept* - Announce successful completion of parsing (input string belongs to the language of the grammar)

4. *Error* - Discover a syntax error and call an error recovery routine.

Shift-Reduce Parsing (3)

- We use **$** to mark the bottom of the stack and also the right end of the input.

STACK	INPUT
$	w $

- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.
- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	INPUT
$ S	$

Shift-Reduce Parsing (4)

Grammar

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

STACK	INPUT	ACTION
\$	$\mathbf{id}_1 * \mathbf{id}_2$ \$	shift
\$ $\mathbf{id}_1$	$* \mathbf{id}_2$ \$	reduce by $F \rightarrow \mathbf{id}$
\$ F	$* \mathbf{id}_2$ \$	reduce by $T \rightarrow F$
\$ T	$* \mathbf{id}_2$ \$	shift
\$ $T *$	$\mathbf{id}_2$ \$	shift
\$ $T * \mathbf{id}_2$	\$	reduce by $F \rightarrow \mathbf{id}$
\$ $T * F$	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept

Figure 4.28: Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

Conflicts During Shift-Reduce Parsing

- Two types of conflicts arises in shift-reduce parsing:

  1. *A shift/reduce conflict – occurs if the parser has a choice to select both shift action and reduce action simultaneously.*

Grammar

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

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \rightarrow \mathbf{id}$
$ $T * F$	$	reduce by $T \rightarrow T * F$
$ T	$	reduce by $E \rightarrow T$
$ E	$	accept

Conflicts During Shift-Reduce Parsing

- Two types of conflicts arises in shift-reduce parsing:

  2. *A reduce/reduce conflict – occurs if more than one reduction is possible for the corresponding handles.*

Grammar

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

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \rightarrow \mathbf{id}$
→ $ $T * F$	$	reduce by $T \rightarrow T * F$
$ T	$	reduce by $E \rightarrow T$
$ E	$	accept

Class Assignment

Q1 Generate the shift-reduce parser for the input string "**(a , (a , a))**", with the help of following grammar.

Grammar (Q1)

$$S \rightarrow (L) \mid a$$
$$L \rightarrow L,S \mid S$$

Q2 Generate the shift-reduce parser for the input string "$a_1 - (a_2 + a_3)$", with the help of following grammar.

Grammar (Q2)

$$S \rightarrow S + S$$
$$S \rightarrow S - S$$
$$S \rightarrow (S)$$
$$S \rightarrow a$$

LR Parsing

- Handwritten notes and online tutorial

LR Parsing

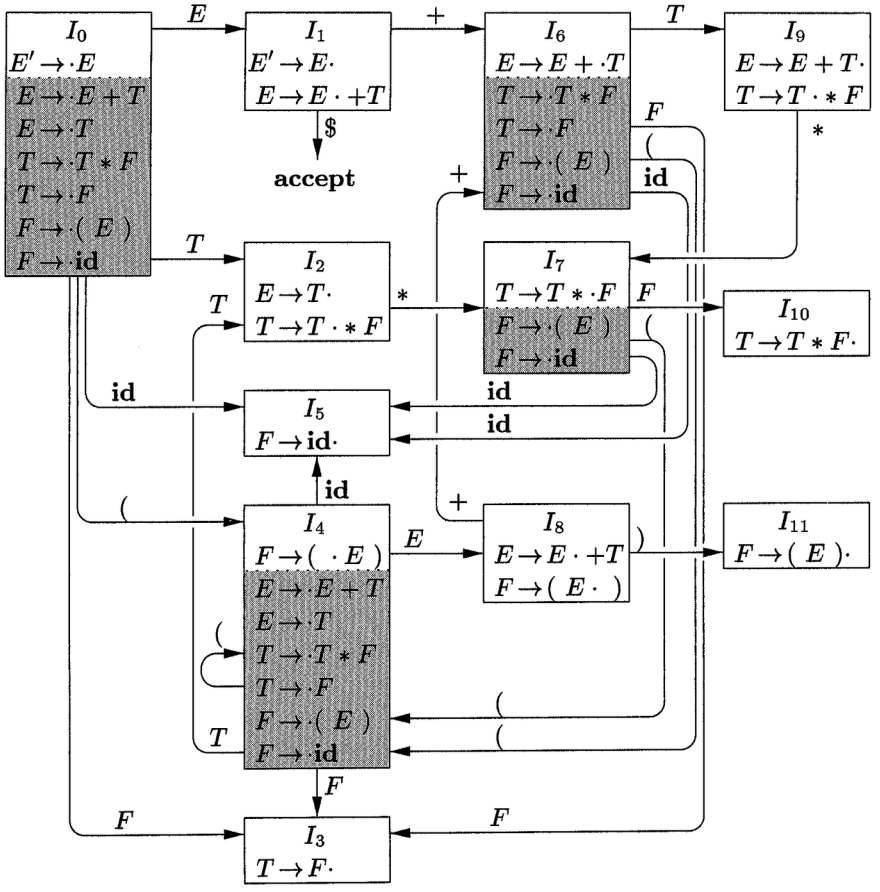


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

LR Parsing

STATE	ACTION						GOTO		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
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			

Figure 4.37: Parsing table for expression grammar

References

Reference for this topic

- **Book:** Alfred V. Aho, Monica S. Lam, Ravi Sethi, J D Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd Edition, Prentice Hall, 2006.
- **Web:** *CS143 Compilers*, Lecture 4, Stanford University.
  https://cs.nyu.edu/courses/Fall12/CSCI-GA.2130-001/lecture4.pdf
- **Web:** Theory of Computation by Ms. Vandita Grover, Assistant Professor, ANDC, University of Delhi
  https://sites.google.com/view/courses-vanditagrover/home/ToC-Resources
- **Web:** ***Java T Point***, Compiler Tutorial
  https://www.javatpoint.com/derivation
  https://www.javatpoint.com/parse-tree