Programming Languages – Lexical and Syntax Analysis

Jongwoo Lim

Introduction

- Language implementation systems must analyze source code, regardless of the specific implementation approach.
- Nearly all syntax analysis is based on a formal description of the syntax of the source language (BNF).
  - Compared to informal syntax descriptions, BNF descriptions are
    - Clear and concise.
    - Used as the direct basis for the syntax analyzer.
    - Relatively easy to maintain due to their modularity.

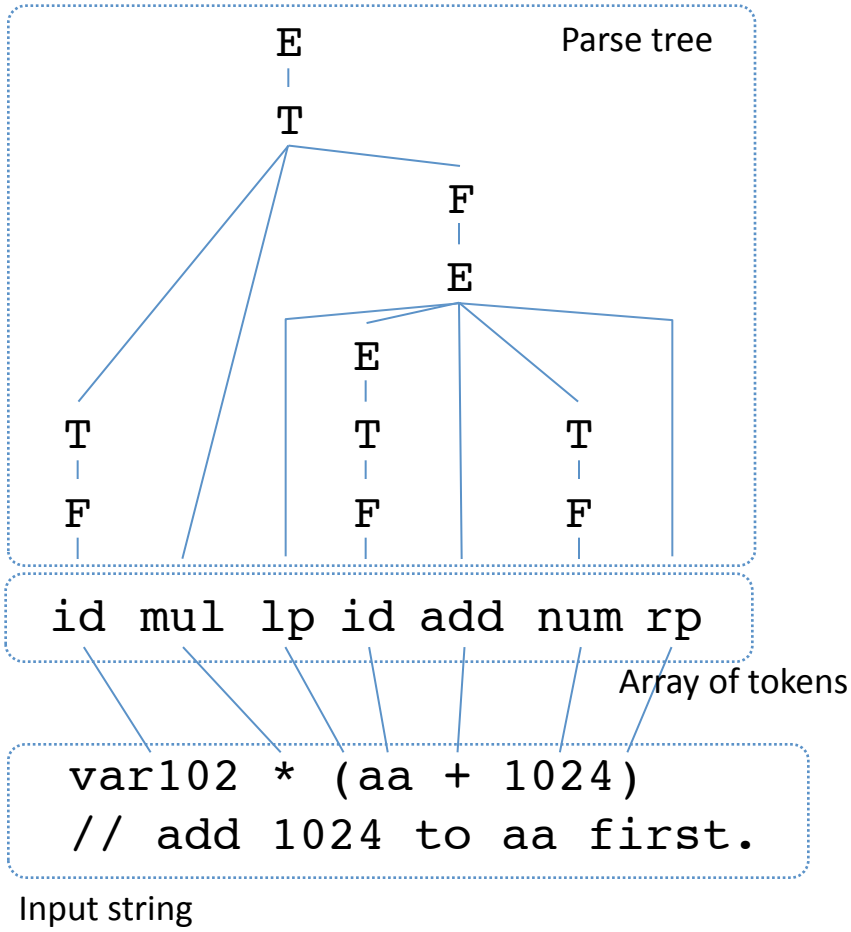
Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a **lexical analyzer** (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a **syntax analyzer**, or **parser** (mathematically, a push-down automaton based on a context-free grammar, or BNF)
- Why?
  - Simplicity: less complex approaches can be used for lexical analysis; separating them simplifies the parser
  - Efficiency: separation allows optimization of the lexical analyzer
  - Portability: parts of the lexical analyzer may not be portable, but the parser always is portable

Syntax Analysis Example

1. E → E add T
2. E → T
3. T → T mul F
4. T → F
5. F → lp E rp
6. F → id | num

```
digit : [0-9]
alpha : [a-zA-Z]
id : alpha(alpha|digit)*
num : digit+
    |(digit+ '.' digit+)
add : '+'
mul : '*'
lp : '('
rp : ')'
```



Lexical Analysis

- A lexical analyzer is a pattern matcher for character strings.
- A lexical analyzer is a “front-end” for the parser.
- Identifies substrings of the source program that belong together - **lexemes**
  - Lexemes match a character pattern, which is associated with a lexical category called a token.

    ```
    sum = a + b;
    ```

  - `sum` is a lexeme; its token may be `IDENT`.
  - = is a lexeme and its token may be `ASSIGN_OP`.
  - Skip comments and blanks outside lexemes.
  - Put user-defined names into the symbol table.
  - Detect syntactic errors in tokens (e.g. 0.0001.23, 2to3).

Lexical Analyzer

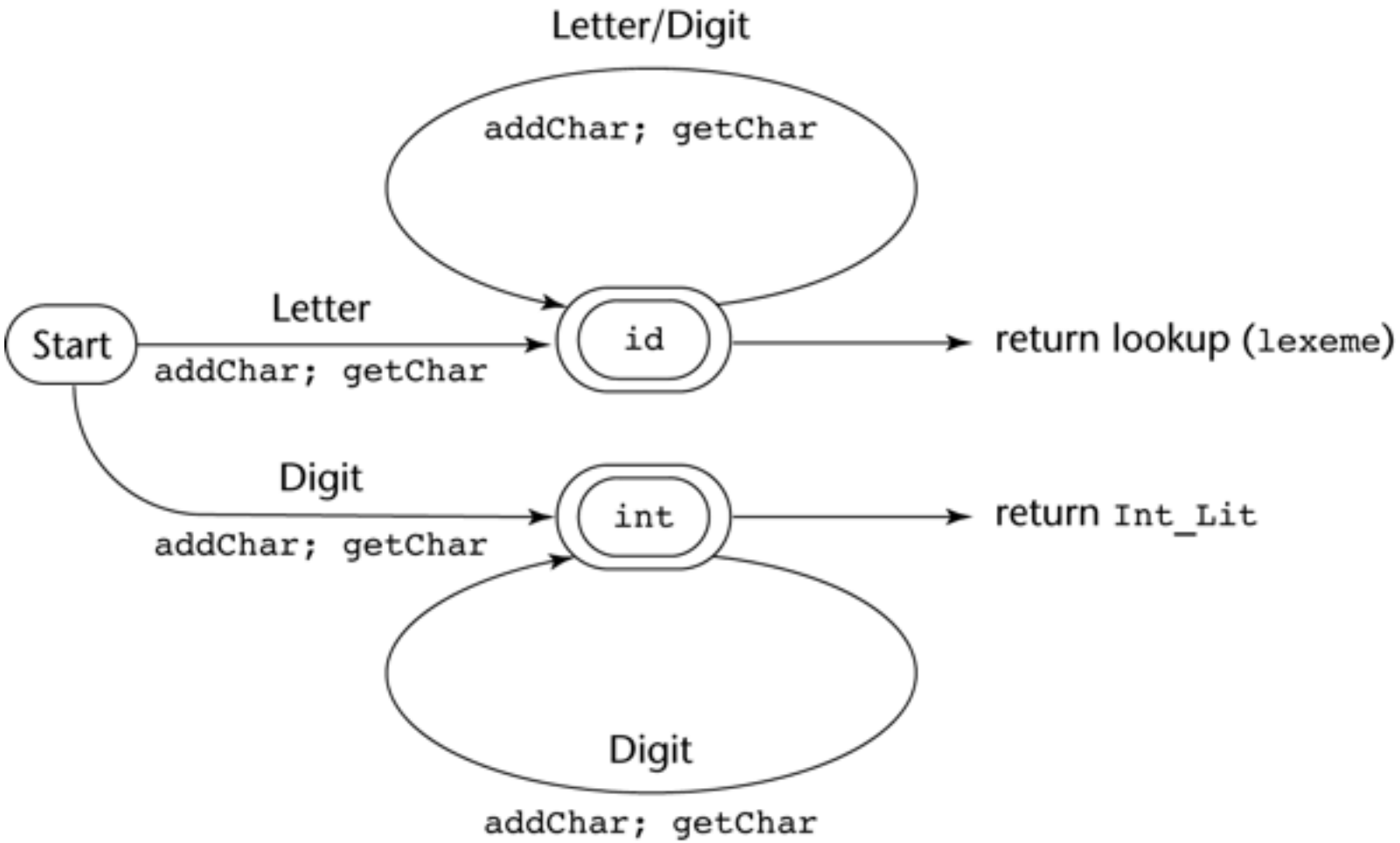
- The lexical analyzer is usually a function that is called by the parser when it needs the next token.
- Three approaches to building a lexical analyzer:
  - Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyzers given such a description.
  - Design a **state diagram** that describes the tokens and write a program that implements the state diagram.
  - Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram.

State Diagram Design

- State transition diagram:
  - A directed graph, whose nodes are state names, and whose edges(arcs) are with input characters that cause the transition.
  - Mathematically, a finite automaton based on a regular grammar.
  - Example of regular grammars: wildcard in filename matching.

```
*.txt, test[0-9]+.doc, …

*.txt : .txt, b.txt, abcd.txt 12c4e.txt a_b_123.txt
    but not a.pdf, abc.text, abc.txt.pdf .

test[0-9]+.doc : test0.doc, test1023.doc, test001.doc
    but not test.doc, testabc.doc, test00a.doc .
```

State Diagram



State Diagram Design

- State transition diagram
  - A naïve state diagram would have a transition from every state on every character in the source language - such a diagram would be very large!
  - Group characters in the same transition.
    - e.g. `LETTER` for all 52 upper- and lower-case characters, a digit class `DIGIT` for all digits.
  - Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word).
    - Use a table lookup to determine whether a possible identifier is in fact a reserved word.

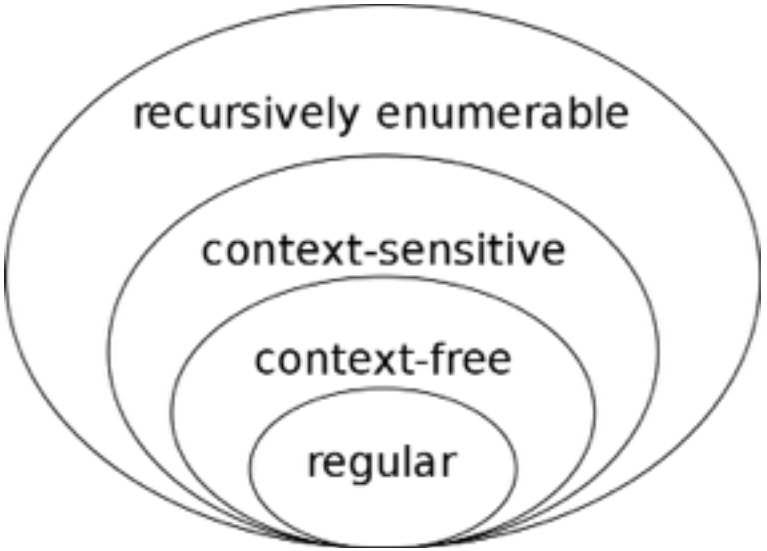
Lexical Analyzer

- Convenient utility subprograms:
  - `getChar` - gets the next character of input, puts it in `nextChar`, determines its class and puts the class in `charClass` .
  - `addChar` - puts the character from `nextChar` into the place the lexeme is being accumulated, `lexeme` .
  - `lookup` - determines whether the string in `lexeme` is a reserved word (returns a code).
- Not a good example!
  - Avoid using global variables.
  - Function names do not show there meanings clearly.
  - Complex structure.

Chomsky Hierarchy

Grammar	Languages	Automaton	Production rules
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$ (no restriction)
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow a$ and $A \rightarrow aB$

From wikipedia



Regular Language

- Formal definition:
  - The empty language $\emptyset$ is a regular language.
  - For each $a \in \Sigma$, the singleton language $\{ a \}$ is a regular language.
  - If A and B are regular languages, then
    - $A \cup B$ (union, OR),
    - $A \cdot B$ (concatenation), and
    - $A^\star$ (Kleene star) are regular languages.
  - No other languages over Σ are regular.
- Accepted by a finite state machine.

Regular Expressions

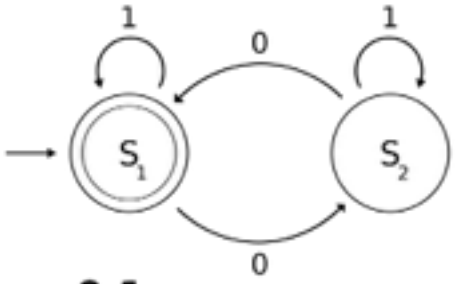
- Boolean OR: ‘gray’ or ‘grey’ → gray|grey
- Grouping: gr(a|e)y
- Quantification: repetition (?, *, +), e.g. ab*c → ac, abc, abbc, …

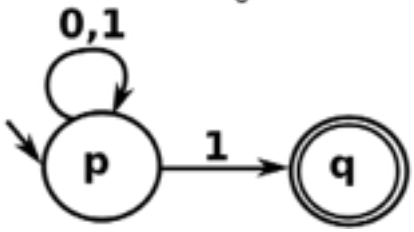
POSIX (extended) regular expressions		
`.`	Matches any single character (many applications exclude newlines).	`a.c [a.c]`
`[ ]`	Matches a single character that is contained within the brackets.	`[abc] [a-z] [abcx-z]`
`[^ ]`	Matches a single character that is not contained within the brackets.	`[^abc] [^a-z]`
`^`	Matches the starting position within the string.	`^abc`
`$`	Matches the ending position of the string.	`abc$`
`( )`	Defines a marked subexpression (a block or capturing group).	`a(b|c)d`
`|`	The choice (aka alternation or set union) operator.	`abc|def`
`*`	Matches the preceding element zero or more times.	`ab*c [xyz]* (ab)*`
`?`	Matches the preceding element zero or one time.	`ab? a.?`
`+`	Matches the preceding element one or more times.	`ab+`
`{m,n}`	Matches the preceding element at least *m* and not more than *n* times.	`a{3,5}`

Finite-state Automaton

- Finite-state machine
  - It is in only one state at a time (the current state).
  - It changes state by a triggering event or condition (transition).
  - Acceptors (recognizers): start and accept (final) state.
  - Deterministic vs. non-deterministic.
    - Multiple transitions for the same symbol.
    - Transitions without input symbols ("ε-moves").

State transition diagrams



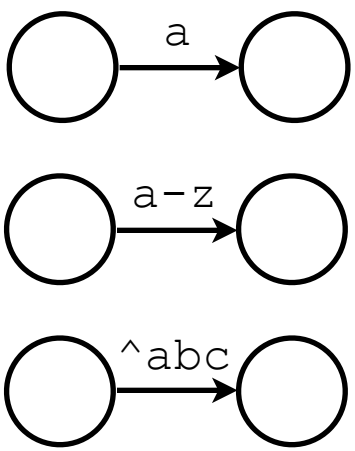


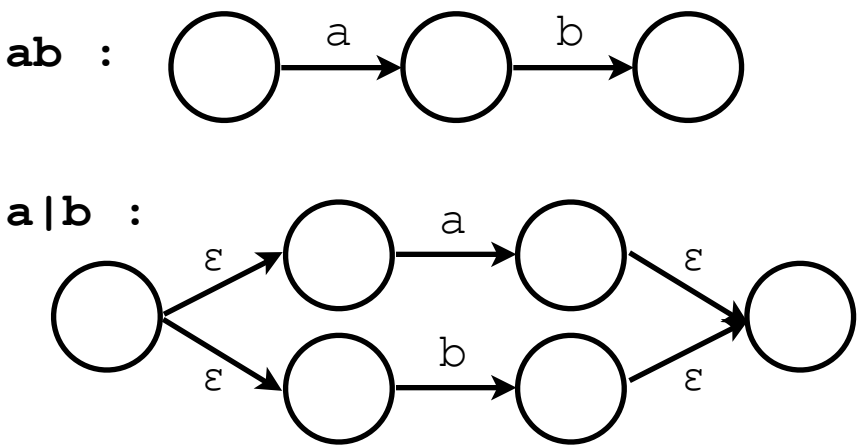
	0	1
S	S	S
S	S	S

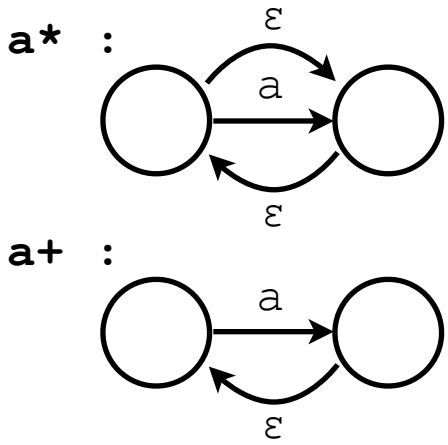
State/event tables

	0	1
p	p	p,q
q	-	-

Regular Expressions



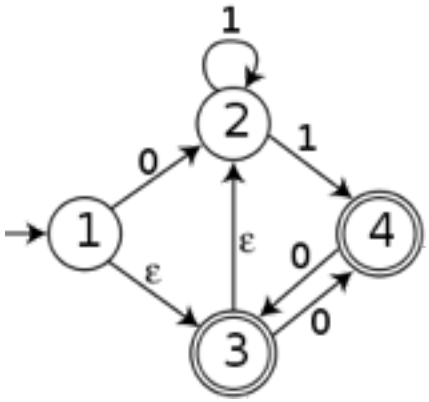




POSIX (extended) regular expressions		
`.`	Matches any single character (many applications exclude newlines).	`a.c [a.c]`
`[ ]`	Matches a single character that is contained within the brackets.	`[abc] [a-z] [abcx-z]`
`[^ ]`	Matches a single character that is not contained within the brackets.	`[^abc] [^a-z]`
`^`	Matches the starting position within the string.	`^abc`
`$`	Matches the ending position of the string.	`abc$`
`( )`	Defines a marked subexpression (a block or capturing group).	`a(b|c)d`
`|`	The choice (aka alternation or set union) operator.	`abc|def`
`*`	Matches the preceding element zero or more times.	`ab*c [xyz]* (ab)*`
`+`	Matches the preceding element one or more times.	`ab+`

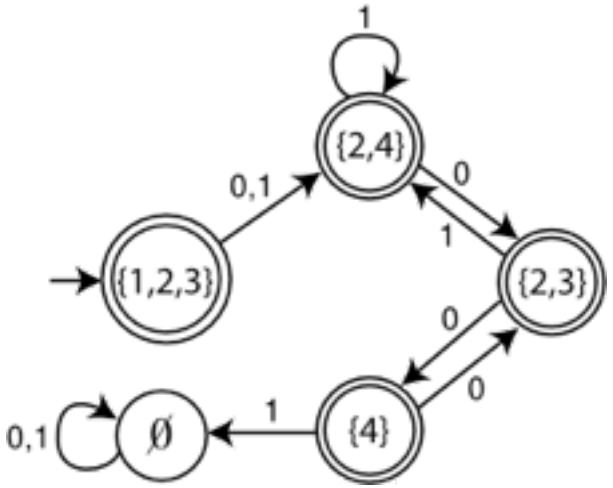
Nondeterministic Finite-state Automaton

- Powerset construction:
  - Converting a nondeterministic finite-state automaton (NFA) to a deterministic one (DFA).
  - If the NFA has n states, the resulting DFA may have 2^n states.



	ε	0	1
1	3	2	-
2	-	-	2,4
3	2	4	-
4	-	3	-

	0	1
{1,2,3}	{2,4}	{2,4}
{2,4}	{2,3}	{2,4}
{2,3}	{4}	{2,4}
{4}	{2,3}	ø
ø	ø	ø



Nondeterministic Finite-state Automaton

- Powerset construction:
  - Initial state of the resulting DFA:
    - The initial state of NFA, and
      the states reachable by ε-moves from the initial state.
  - For each state in the DFA whose transition is unknown:
    - For each symbol,
      find the states in NFA after the transition by the symbol, and
      all the states reachable by ε-moves from those states.
    - Repeat until no such states in the DFA exist.

Lex / Flex

- Lexical analyzer generator.

```
%{
#include <stdio.h>
%}
...
digit          [0-9]
letter         [a-zA-Z]

%% /*** Rules section ***/

"var"      { printf("VAR\n"); }
"while"    { printf("WHILE\n"); }
{letter}({letter}|{digit})*  { printf("ID: %s\n", yytext); }
{digit}+   { printf("Integer: %s\n", yytext); }
.|\n       { /* Ignore all other characters. */ }

%% /*** C Code section ***/

int main(void) {
  yylex();
  return 0;
}
```

The Parsing Problem

- Goals of the parser:
  - Given an input program, find all syntax errors;
    for each, produce an appropriate diagnostic message and recover quickly.
    - Detect as many errors as possible during a single analysis.
  - Produce the parse tree, or at least a trace of the parse tree, for the program.

The Parsing Problem

- Two categories of parsers
  - Top down: produce the parse tree, beginning at the root.
    - Order is that of a leftmost derivation.
    - Traces or builds the parse tree in preorder.
  - Bottom up: produce the parse tree, beginning at the leaves.
    - Order is that of the reverse of a rightmost derivation.
- Useful parsers look only one token ahead in the input.

Notational conventions in the chapter.		
Terminal symbols	Lowercase letters at the beginning of alphabet.	a, b, c, …
Non-terminal symbols	Uppercase letters at the beginning of alphabet.	A, B, C, …
Terminals OR non-terminals	Uppercase letters at the end of alphabet.	Z, Y, X, W, …
Strings of terminals	Lowercase letters at the end of alphabet.	z, y, x, w, …
Mixed strings	Lowercase Greek letters.	α, β, γ, δ, …

Top-down Parsers

- Top-down Parsers
  - Builds a parse tree in preorder (leftmost derivation).
  - Given a sentential form, xAα, and the A-rules,

    e.g. A → bB, A → cBb, and A → a,

    the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation,

    e.g. xbBα, xcBbα, or xaα,

    (using only the first token produced by A).
- The most common top-down parsing algorithms:
  - Recursive-descent parser: a coded implementation of BNF.
  - LL parsers: table driven implementation.
    - Left-to-right scan, and a Leftmost derivation is generated.

Bottom-up Parsers

- Bottom-up parsers
  - Builds a parse tree from leaves toward the root – the reverse of a rightmost derivation.
  - Given a right sentential form, α, determine what substring of α is the right-hand side of the rule (**handle**) in the grammar that must be reduced to produce the previous sentential form in the right derivation.

    e.g. $S \rightarrow aAc$, $A \rightarrow aA \mid b$, and

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

    then $aa\underline{b}c >> a\underline{aA}c >> \underline{aAc} >> S$.
  - The most common bottom-up parsers are in the LR family.
    - Left-to-right scan, and a Rightmost derivation is generated.

The Parsing Problem

- The Complexity of Parsing
  - Parsers that work for any unambiguous grammar are complex and inefficient – $O(n^3)$, where n is the length of the input.
  - Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time – O(n).

Recursive-Descent Parsing

- There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal.
- EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals.

Recursive-Descent Parsing

- Assume we have a lexical analyzer named lex, which puts the next token code in `nextToken`.
  - Every parsing routine leaves the next token in `nextToken`.
- The coding process when there is only one RHS:
  - For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error.
  - For each nonterminal symbol in the RHS, call its associated parsing subprogram.

Recursive-Descent Parsing

- A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse.
  - The correct RHS is chosen on the basis of the next token of input (the lookahead).
  - The next token is compared with the first token that can be generated by each RHS until a match is found.
  - If no match is found, it is a syntax error.

Recursive-Descent Parsing

- A grammar for simple expressions:

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

Recursive Descent Parsing

<expr> → <term> {(+ | -) <term>}

```
/* Function expr
   Parses strings in the language generated by the rule:
   <expr> → <term> {(+ | -) <term>}
 */

void expr() {
  printf("Enter <expr>\n");
  term();  // First term.

  // As long as the next token is + or -, call lex() to get
  // the next token and parse the next term.
  while (nextToken == ADD_OP || nextToken == SUB_OP){
    lex();
    term();
  }
  // This routine does not detect errors.
  printf("Exit <expr>\n");
}
```

Recursive Descent Parsing

<term> → <factor> {(* | /) <factor>}

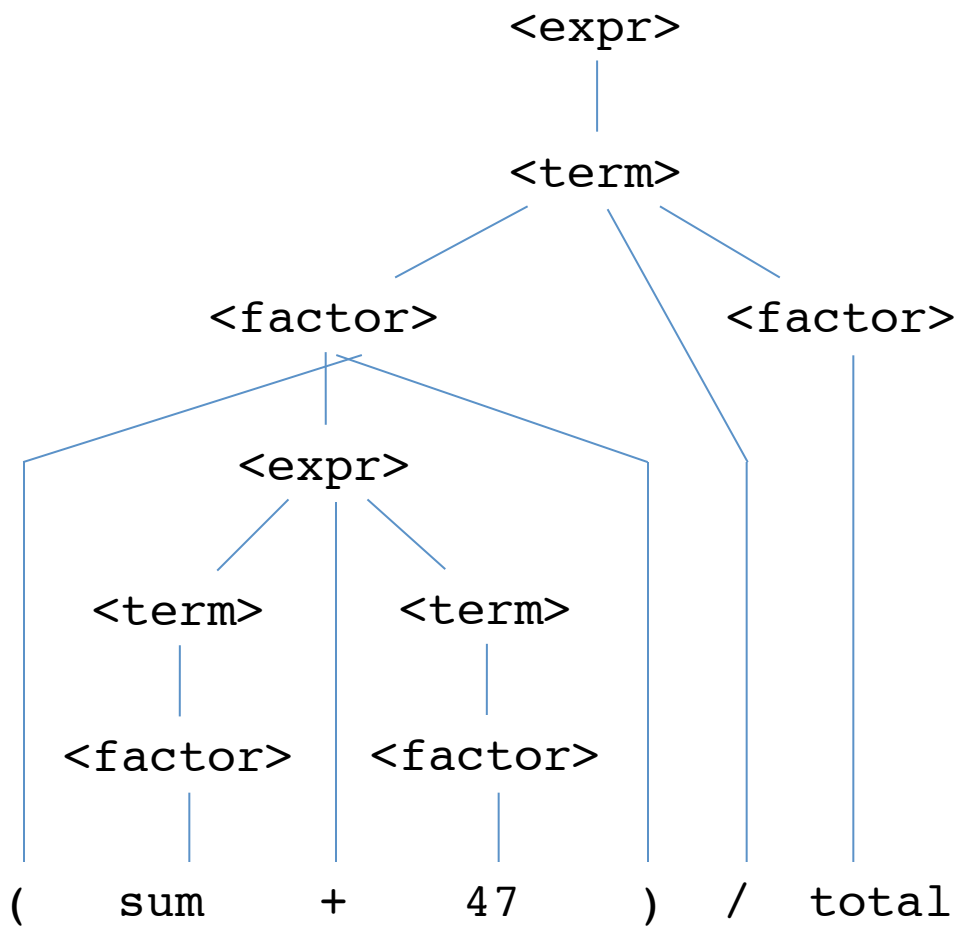
```
/* Function term
   Parses strings in the language generated by the rule:
   <term> -> <factor> {(* | /) <factor>)
 */

void term() {
  printf("Enter <term>\n");
  factor();  // Parse the first factor.

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

Recursive Descent Parsing

<factor> → id | int_constant | (<expr>)

```
/* Function factor
   Parses strings in the language generated by the rule:
   <factor> -> id | (<expr>)
*/

void factor() {
  // Determine which RHS.
  if (nextToken) == ID_CODE || nextToken == INT_CODE) {
    // For the RHS id or int_constant, just call lex.
    lex();
  } else if (nextToken == LP_CODE) {
    // If the RHS is (<expr>) – call lex to pass over '(',
    // call expr, and check for ')'.
    lex();
    expr();
    if (nextToken == RP_CODE) lex();
    else error();
  } else {
    error();  // Neither RHS matches.
  }
}
```

Recursive Descent Parsing



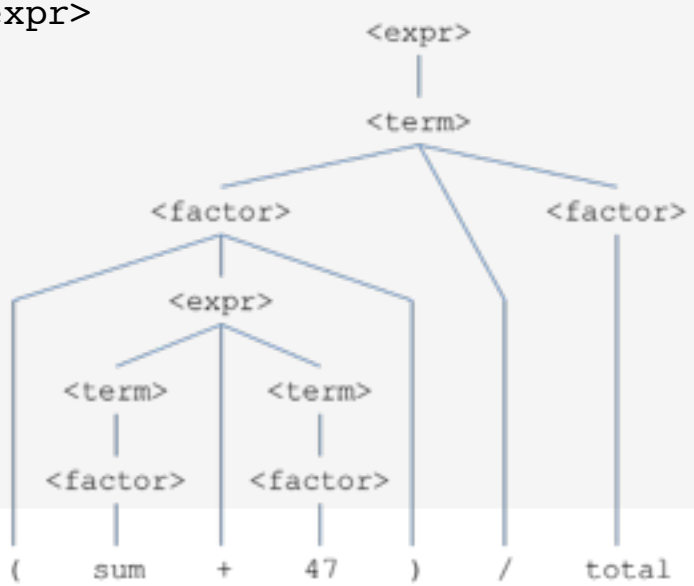

```
void expr() {
  printf("Enter <expr>\n");
  term();
  while (nextToken == ADD_OP ||
         nextToken == SUB_OP){
    lex();
    term();
  }
  printf("Exit <expr>\n");
}

void term() {
  printf("Enter <term>\n");
  factor();
  while (nextToken == MULT_OP ||
         nextToken == DIV_OP) {
    lex();
    factor();
  }
  printf("Exit <term>\n");
}

void factor() {
  if (nextToken) == ID_CODE ||
      nextToken == INT_CODE) {
    lex();
  } else if (nextToken == LP_CODE) {
    lex();
    expr();
    if (nextToken == RP_CODE) lex();
    else error();
  } else {
    error();
  }
}
```

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>
```



Recursive Descent Parsing

<if_stmt> → if (<bool_expr>) <stmt> [else <stmt>]

```
void if_stmt() {
  if (nextToken != IF_CODE) {
    error();
  } else {
    lex();
    if (nextToken != LEFT_PAREN) {
      error();
    } else {
      lex();
      bool_expr();
      if (nextToken != RIGHT_PAREN) {
        error();
      } else {
        lex();
        stmt();
        if (nextToken == ELSE_CODE) {
          lex();
          stmt();
        }
      }
    }
  }
}
```

LL Grammar Class

- LL(k) parser
  - LL parser that requires k lookahead tokens.
  - We only deal with LL(1) parser in this class.
- Example : parsing using an LL(1) parser.

Example: (a + a)

token	rule	stack
		S, EOF
(	2	(, S, +, F,), EOF
(		S, +, F,), EOF
a	1	F, +, F,), EOF
a	3	a, +, F,), EOF
a		+, F,), EOF
+		F,), EOF
a	3	a,), EOF
a		), EOF
)		EOF

1. S → F
2. S → (S + F)
3. F → a

	(	)	a	+	EOF
S	2	-	1	-	-
F	-	-	3	-	-

Left Recursion

- The Left Recursion Problem
  - If a grammar has **left recursion**, either direct or indirect, it cannot be the basis for a top-down parser.
  - A grammar can be modified to remove left recursion:
    For each nonterminal, A,
    Group the A-rules as $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$,
    where none of the β's begins with A.
    Replace the original A-rules with
    $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$
    $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

Disjointness

- The other characteristic of grammars that disallows top-down parsing is the lack of pairwise disjointness.
  - The inability to determine the correct RHS on the basis of one token of lookahead.
  - Definition the **first-set**: FIRST(α) = { c | α ⇒* c β }

    If α ⇒* ε , ε is in FIRST(α).

- Pairwise Disjointness Test:
  - For each nonterminal, A, that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$, and $A \rightarrow \alpha_j$, $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varnothing$.

    Examples:

    A → aB | bAb | Bb, B → cB | d FIRST of A-rules: {a}, {b}, {c,d}

    A → aB | Bab, B → aB | b FIRST of A-rules: {a}, {a,b}

Left Factoring

- Left factoring can resolve the problem
  - Leave the common parts and separate the rest into a new rule.
  - Example:

    <var> → identifier | identifier ‘[’ <expression> ‘]’

    can be written as

    <var> → identifier <new>

    <new> → ε | ‘[’ <expression> ‘]’

    or

    <var> → identifier [‘[’ <expression> ‘]’]

    (the outer brackets are meta-symbols of EBNF)

First Set

- Build the first-set.
  - Given a grammar with the rules $A_1 \rightarrow \omega_1$, ... , $A_n \rightarrow \omega_n$,
  - Define FIRST(X) for a single terminal or nonterminal X:
    - X is a terminal 'a': FIRST(X) = {a}.
    - X is 'ε': FIRST(X) = {ε}.
    - X is a nonterminal: for every rule $X \rightarrow B_1 B_2 \ldots B_m$,

      Put $FIRST(B_1) - \{\varepsilon\}$ into FIRST(X).

      If all $FIRST(B_j)$ contains ε, $1 \le j < i$, $i \le m$,

      put $FIRST(B_i) - \{\varepsilon\}$ into FIRST(X).

      If all $FIRST(B_i)$ contains ε , $1 \le i \le m$, put {ε} into FIRST(X).

First Set

- Algorithm:
  - Given a grammar with the rules $A_1 \rightarrow \omega_1$, ... , $A_n \rightarrow \omega_n$,
  - Initialize FIRST(A) $\leftarrow$ { }, $\forall$A.
  - Iterate until no changes in all FIRST(A) :
    - For each rule $A \rightarrow X_1 X_2 \ldots X_m$, set k = 1, empty = true.
    - While empty == true && $k \leq m$,

      Put FIRST(X_k) − {ε} into FIRST(A).

      empty = ($\varepsilon \in$ FIRST(X_k)), k = k + 1.
    - If empty == true , put {ε} into FIRST(A).

First Set Examples

S → F
S → (S + F)
F → a

Iterations

	1	2
S	(	(, a
F	a	

E → E + T | T
T → T * F | F
F → (E) | id

Iterations

	1	2	3
E	∅	∅	(, id
T	∅	(, id	
F	(, id		

E → T E′
E′ → + T E′ | ε
T → F T′
T′ → * F T′ | ε
F → (E) | id

Iterations

	1	2	3
E	∅	∅	(, id
E′	+, ε		
T	∅	(, id	
T′	*, ε		
F	(, id		

FirstSet Algorithm

```
Initialize FIRST(A) ← {}, ∀A.
Iterate until no changes in all FIRST(A) :
  For each rule A
  Set k = 1, empty = true.
  While empty == true && k ≤ m,
    Put FIRST(X
    k = k + 1, empty = (
  If empty == true, put {ε} into FIRST(A).
```

Follow Set

- Follow set for a nonterminal *A*:
  - Set of terminals that can appear at the immediately right of *A*.
    e.g. 'a' is in FOLLOW(*A*) if $B \Rightarrow^+ \beta A a \gamma$.
  - If *A* can be at the rightmost symbol, $ (EOF) is in FOLLOW(*A*).
  - 'ε' can never be in follow sets.
- Build follow sets for the nonterminals *A*.
  - If *A* is the start state, put $ into FOLLOW(*A*).
  - For each rule $B \rightarrow \alpha A \beta$, put FIRST(β) − {ε} into FOLLOW(*A*).
  - If FIRST(β) has ε, put FOLLOW(*B*) to FOLLOW(*A*).
  - For each rule $C \rightarrow \gamma A$, put FOLLOW(*C*) to FOLLOW(*A*).

LL(1) Parser

- To build the parsing table,
  - For each rule $A \rightarrow \alpha$,
  - For each terminal 'a' in FIRST(α), put α in Table(*A*, a).
    - If ε is in FIRST(α),

      For each terminal 'a' in FOLLOW(*A*), put α in Table(*A*, a).
- The grammar is not LL(1) if and only if

  there is more than one entry for any cell in the table.

LL(1) Parser

- Parsing using the parsing table.

  Example: (a + a)

1. S → F
2. S → (S + F)
3. F → a

token	rule	stack
		S, EOF
(	2	(, S, +, F,), EOF
(		S, +, F,), EOF
a	1	F, +, F,), EOF
a	3	a, +, F,), EOF
a		+, F,), EOF
+		F,), EOF
a	3	a,), EOF
a		), EOF
)		EOF

	(	)	a	+	EOF
S	2	-	1	-	-
F	-	-	3	-	-

LL(1) Parser

```
1. S → F :     FIRST(F) = { a }
2. S → (S+F): FIRST((S+F)) = { ( }
3. F → a :     FIRST(a) = { a }
```

	(	)	a	+	$
S	2	-	1	-	-
F	-	-	3	-	-

- Try to parse '`a`', '`(a + a)`', and '`((a + a) + a)`'.

Build LL(1) Parsing Table
For each rule A→α, For each terminal 'a' in FIRST(α), put α in Table(A, a). If ε is in FIRST(α), For each terminal 'a' in FOLLOW(A), put α in Table(A, a).

LL(1) Parser

```
1. S → F S :   FIRST(F S) = { a }
2. S → + S :   FIRST(+ S) = { + }
3. S → ε :     FIRST(ε) = { ε }
               FOLLOW(S) = { $ }
4. F → a :     FIRST(a) = { a }
```

	a	+	$
S	1	2	3
F	4	-	-

- Try to parse 'a', 'a + a', and 'a + a + a'.

Build LL(1) Parsing Table
For each rule A→α, For each terminal 'a' in FIRST(α), put α in Table(A, a). If ε is in FIRST(α), For each terminal 'a' in FOLLOW(A), put α in Table(A, a).

LL(1) Parser

```
1. A → aB :    FIRST(aB) = { a }
2. A → bAb :   FIRST(bAb) = { b }
3. A → Bb :    FIRST(Bb) = { c, d }
4. B → cB :    FIRST(cB) = { c }
5. B → d :     FIRST(d) = { d }
```

	a	b	c	d	$
A	1	2	3	3	-
B	-	-	4	5	-

- Try to parse '`accd`', '`badb`', and '`bbcdbbb`'.

Build LL(1) Parsing Table
For each rule A→α, For each terminal 'a' in FIRST(α), put α in Table(A, a). If ε is in FIRST(α), For each terminal 'a' in FOLLOW(A), put α in Table(A, a).

LL(1) Parser

```
1. A → Ba :   FIRST(Ba) = { b, c }
2. A → CB :   FIRST(CB) = { b, c, d }
3. B → bc :   FIRST(bc) = { b }
4. B → cA :   FIRST(cA) = { c }
5. C → d :    FIRST(d) = { d }
6. C → ε :    FIRST(ε) = { ε }
              FOLLOW(C) = { b, c }
```

	a	b	c	d	$
A	–	1,2	1,2	2	–
B	–	3	4	–	–
C	–	6	6	5	–

- Not an LL(1) grammar.

Build LL(1) Parsing Table

For each rule A→α,
  For each terminal ‘a’ in FIRST(α), put α in Table(A, a).
    If ε is in FIRST(α),
    For each terminal ‘a’ in FOLLOW(A), put α in Table(A, a).

Bottom-up Parsing

- The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation.
  - Bottom-up parsing: use rightmost derivation.

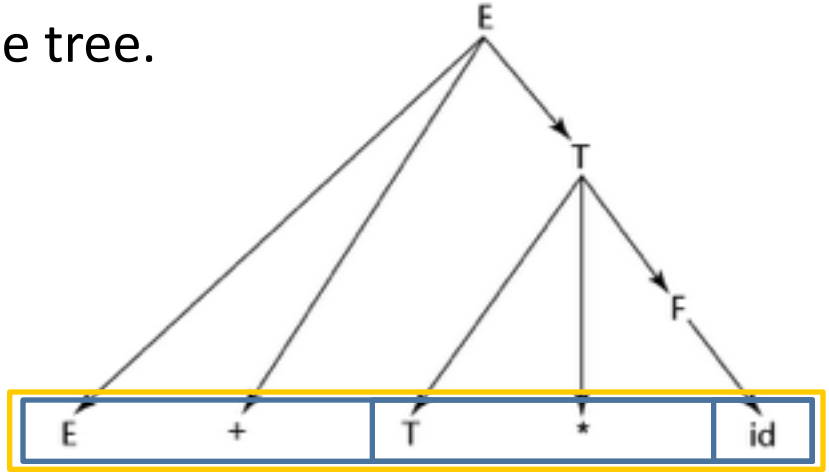
```
id + id * id
E ⇒ E + T
  ⇒ E + T * F
  ⇒ E + T * id
  ⇒ E + F * id
  ⇒ E + id * id
  ⇒ T + id * id
  ⇒ F + id * id
  ⇒ id + id * id
```

```
id + id + id
E ⇒ E + T
  ⇒ E + F
  ⇒ E + id
  ⇒ E + T + id
  ⇒ E + F + id
  ⇒ E + id + id
  ⇒ T + id + id
  ⇒ F + id + id
  ⇒ id + id + id
```

```
E → E + T | T
T → T * F | F
F → ( E ) | id
```

Bottom-up Parsing

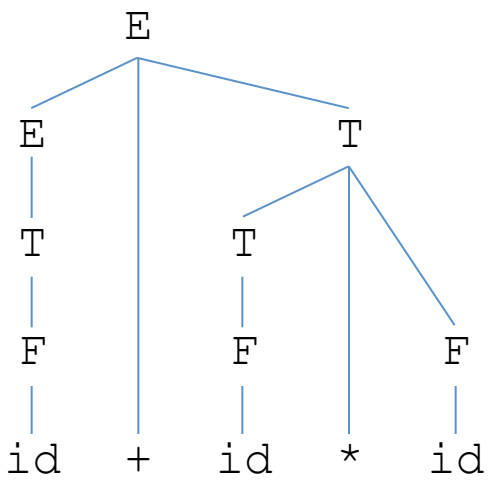
- Intuition about handles.
  - Definition: β is the **handle** of the right sentential form γ=αβw
    if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$.
  - Definition: β is a **phrase** of the right sentential form γ
    if and only if $S \Rightarrow^* \gamma=\alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1\beta\alpha_2$
  - Definition: β is a **simple phrase** of the right sentential form γ
    if and only if $S \Rightarrow^* \gamma=\alpha_1 A \alpha_2 \Rightarrow \alpha_1\beta\alpha_2$.
    - #phrases = #internal nodes in parse tree.
    - A simple phrase is always
      an RHS in the grammar.



Bottom-up Parsing

- Intuition about handles:
  - The handle of any rightmost sentential form is its leftmost simple phrase.
  - Given a parse tree, it is now easy to find the handle.
  - Parsing can be thought of as handle pruning.

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

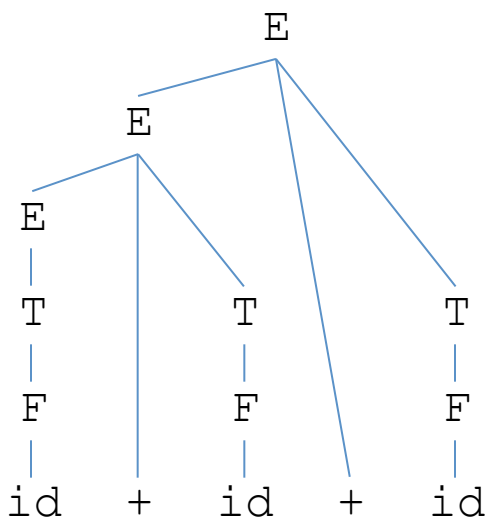


$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid id
\end{aligned}
$$

Bottom-up Parsing

- Intuition about handles:
  - The handle of any rightmost sentential form is its leftmost simple phrase.
  - Given a parse tree, it is now easy to find the handle.
  - Parsing can be thought of as handle pruning.

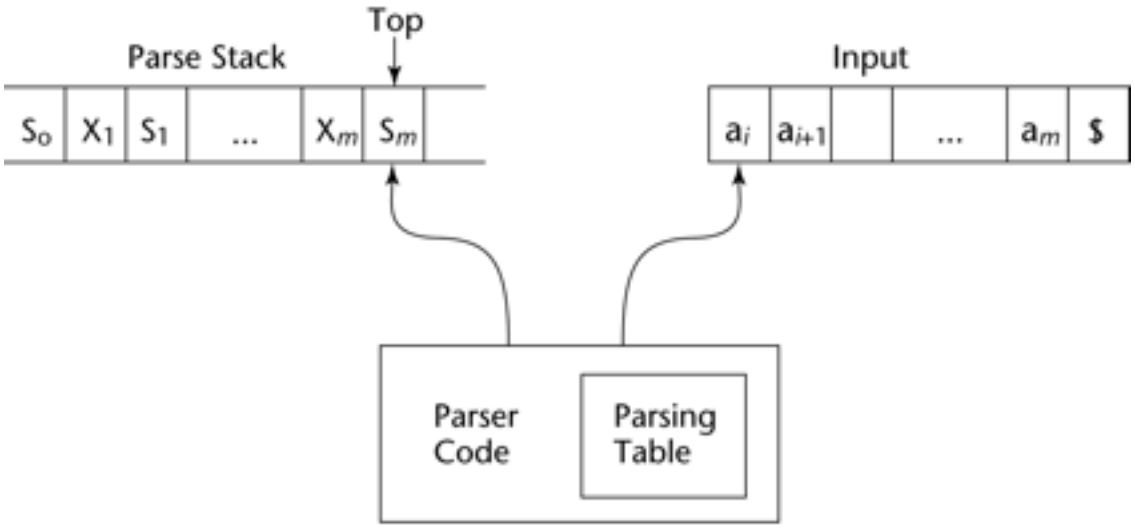
$$
\begin{aligned}
E &\Rightarrow \underline{E + T} \\
&\Rightarrow E + \underline{F} \\
&\Rightarrow E + \underline{id} \\
&\Rightarrow \underline{E + T} + id \\
&\Rightarrow E + \underline{F} + id \\
&\Rightarrow E + \underline{id} + id \\
&\Rightarrow \underline{T} + id + id \\
&\Rightarrow \underline{F}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



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

Shift-Reduce Algorithms

- Shift-Reduce Algorithms:
  - **Shift** action moves the next token onto the parse stack.
  - **Reduce** action replaces the handle on top of the parse stack with its corresponding LHS.
  - Pushdown automaton (PDA).



LR Parsers

- Advantages of LR parsers:
  - They will work for nearly all grammars that describe programming languages.
  - They work on a larger class of grammars than other bottom-up algorithms, but are as efficient as any other bottom-up parser.
  - They can detect syntax errors as soon as it is possible.
  - The LR class of grammars is a superset of the class parsable by LL parsers.
- Disadvantage of LR parsing:
  - The parsing table is hard to produce by hand.
    - Several programs available for this purpose.

LR Parsers

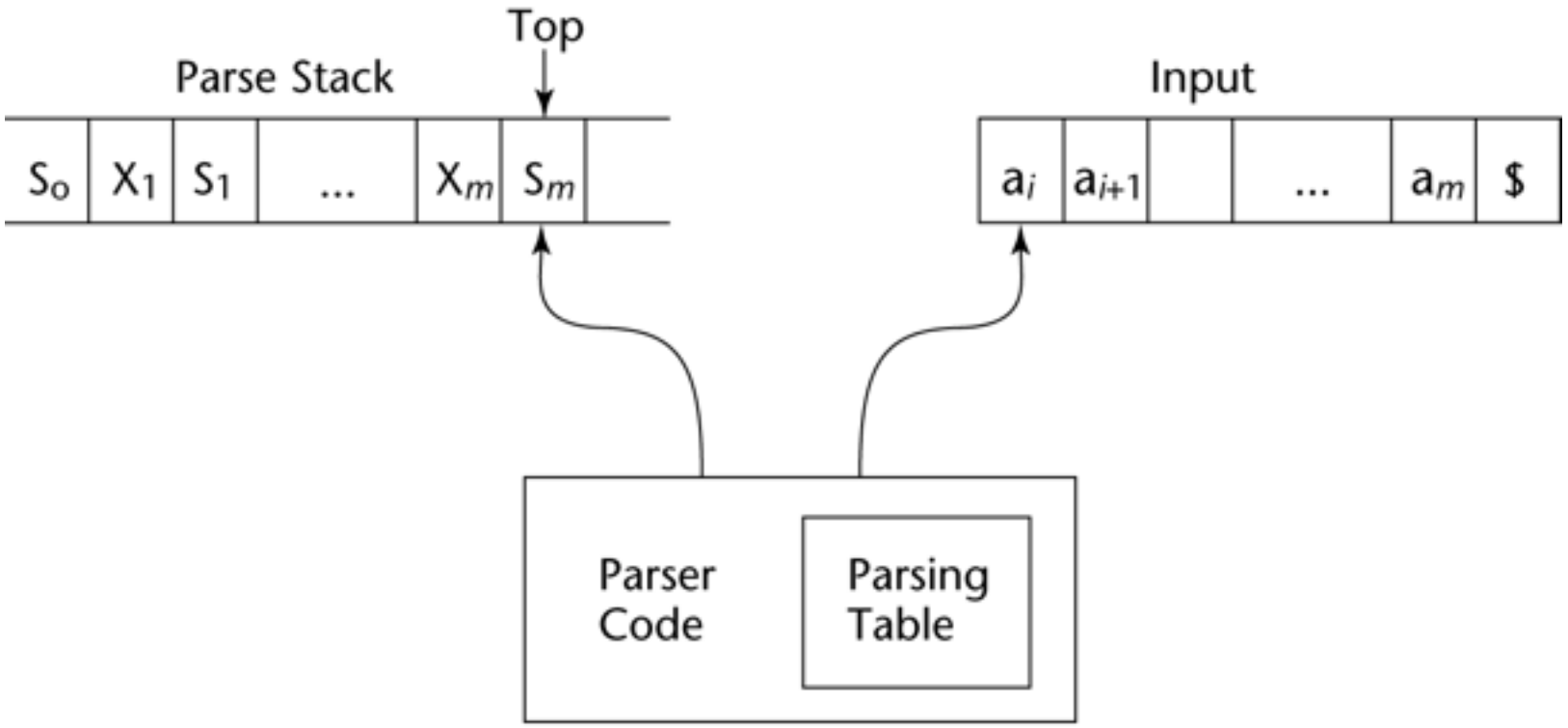
- Knuth's insight:
  - A bottom-up parser could use the entire history of the parse, up to the current point, to make parsing decisions.
  - There were only a finite and relatively small number of different parse situations that could have occurred, so the history could be stored in a parser state, on the parse stack.
  - An LR configuration stores the state of an LR parser

    $(S_0 X_1 S_1 X_2 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$)$
  - S_k : state symbols, X_k : grammar symbols.

Bottom-up Parsing

- An LR configuration stores the state of an LR parser

  $(S_0 X_1 S_1 X_2 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$)$



Bottom-up Parsing

- LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table.
  - The ACTION table specifies the action of the parser, given the parser state and the next token.
    - Rows are state names; columns are terminals.
    - R for reduce and S for shift.
  - The GOTO table specifies which state to put on top of the parse stack after a reduction action is done.
    - Rows are state names; columns are nonterminals.
- A parser table can be generated from a given grammar with a tool, e.g., yacc

LR Parsing Table

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

$(S_0\, a_1\, a_2 \ldots a_n\, \$)$

	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			

Bottom-up Parsing

- Initial configuration: $(S_0, a_1 a_2 \ldots a_n \$)$
  - Parser actions:

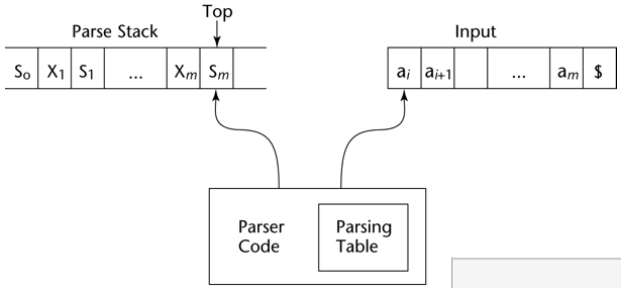
    $(S_0 X_1 S_1 X_2 S_2 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$)$
  - If $ACTION[S_m, a_i]$ = Shift S, the next configuration is:

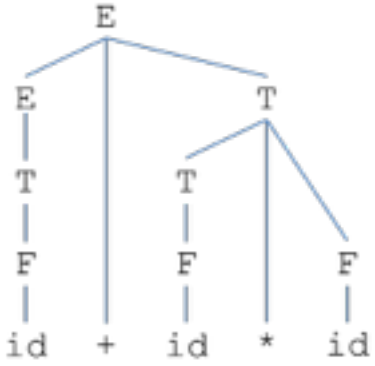
    $(S_0 X_1 S_1 X_2 S_2 \ldots X_m S_m a_i S, a_{i+1} a_{i+2} \ldots a_n \$)$
  - If $ACTION[S_m, a_i]$ = Reduce $A \rightarrow \beta$ and $S = GOTO[S_{m-r}, A]$, where r = the length of β, the next configuration is

    $(S_0 X_1 S_1 X_2 S_2 \ldots X_{m-r} S_{m-r} A S, a_i a_{i+1} \ldots a_n \$)$
  - If $ACTION[S_m, a_i]$ = Accept, the parse is complete and no errors were found.
  - If $ACTION[S_m, a_i]$ = Error, the parser calls an error-handling routine.

LR Parsing Example

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id



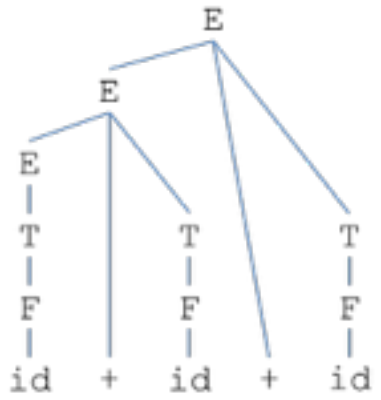


E ⇒ E + T
⇒ E + T * F
⇒ E + T * id
⇒ E + F * id
⇒ E + id * id
⇒ T + id * id
⇒ F + id * id
⇒ id + id * id

	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				**			
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			

Stack	Input	Action
0	id + id * id $	S5
0 id 5	+ id * id $	R6 (G[0,F])
0 F 3	+ id * id $	R4 (G[0,T])
0 T 2	+ id * id $	R2 (G[0,E])
0 E 1	+ id * id $	S6
0 E 1 + 6	id * id $	S5
0 E 1 + 6 id 5	* id $	R6 (G[6,F])
0 E 1 + 6 F 3	* id $	R4 (G[6,T])
0 E 1 + 6 T 9	* id $	S7
0 E 1 + 6 T 9 * 7	id $	S5
0 E 1 + 6 T 9 * 7 id 5	$	R6 (G[7,F])
0 E 1 + 6 T 9 * 7 F 10	$	R3 (G[6,T])
0 E 1 + 6 T 9	$	R1 (G[0,E])
0 E 1	$	Accept

LR Parsing Example



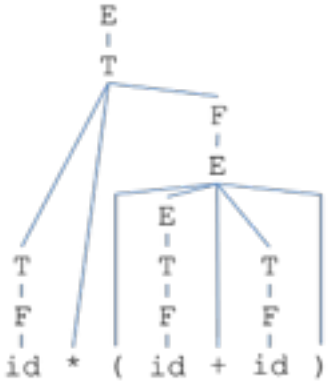
$E \Rightarrow \underline{E + T}$
$\Rightarrow E + \underline{F}$
$\Rightarrow E + \underline{id}$
$\Rightarrow \underline{E + T} + id$
$\Rightarrow E + \underline{F} + id$
$\Rightarrow E + \underline{id} + id$
$\Rightarrow \underline{T} + id + id$
$\Rightarrow \underline{F} + id + id$
$\Rightarrow \underline{id}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

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				**			
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			

Stack	Input	Action
0	id + id + id $	S5
0 id 5	+ id + id $	R6 (G[0,F])
0 F 3	+ id + id $	R4 (G[0,T])
0 T 2	+ id + id $	R2 (G[0,E])
0 E 1	+ id + id $	S6
0 E 1 + 6	id + id $	S5
0 E 1 + 6 id 5	+ id $	R6 (G[6,F])
0 E 1 + 6 F 3	+ id $	R4 (G[6,T])
0 E 1 + 6 T 9	+ id $	R1 (G[0,E])
0 E 1	+ id $	S6
0 E 1 + 6	id $	S5
0 E 1 + 6 id 5	$	R6 (G[6,F])
0 E 1 + 6 F 3	$	R4 (G[6,T])
0 E 1 + 6 T 9	$	R1 (G[0,E])
0 E 1	$	Accept

LR Parsing Example



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

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				**			
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			

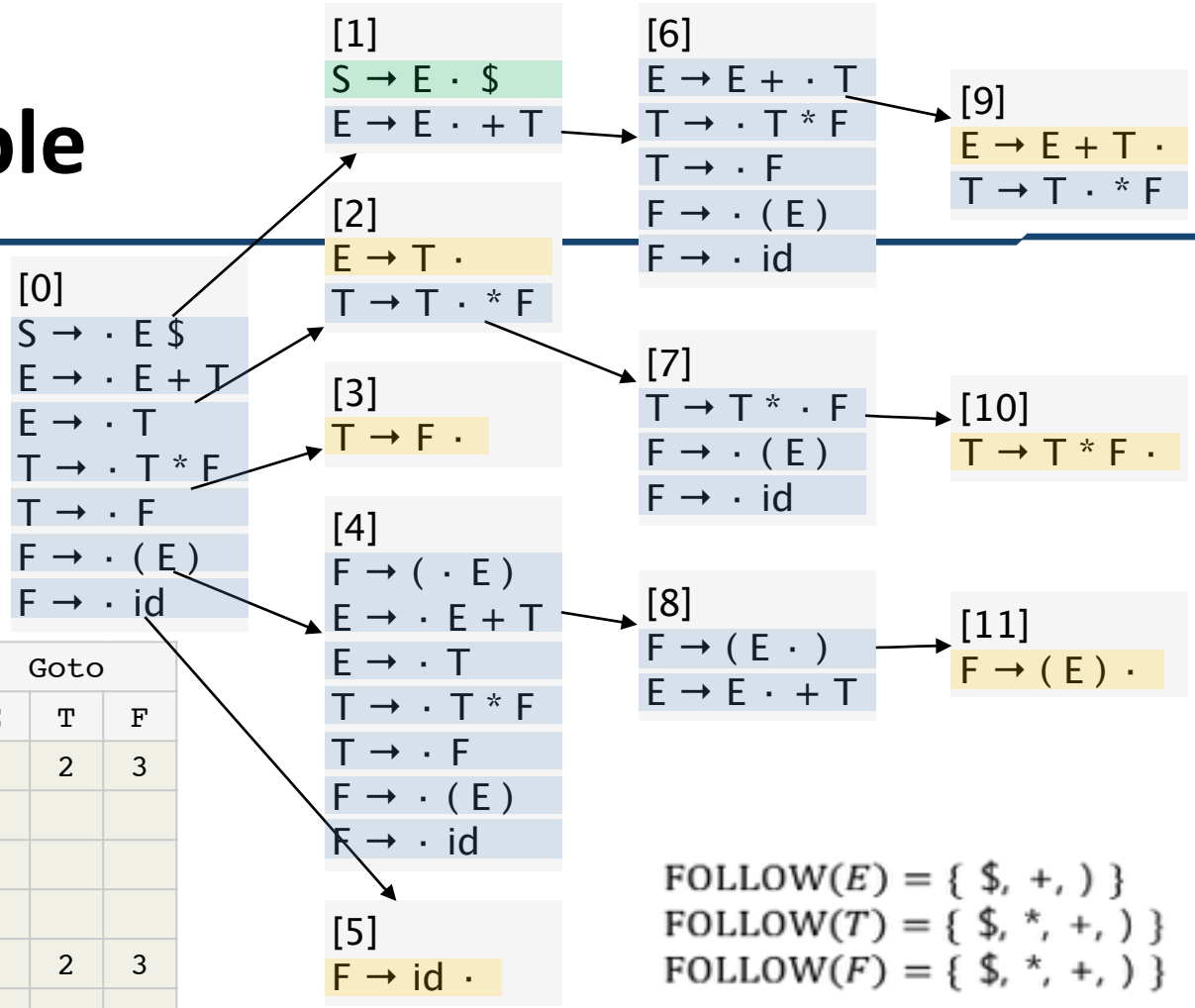
Stack	Input	Action
0	id * (id + id) $	S5
0 id 5	* (id + id) $	R6 (G[0,F])
0 F 3	* (id + id) $	R4 (G[0,T])
0 T 2	* (id + id) $	S7
0 T 2 * 7	(id + id) $	S4
0 T 2 * 7 (4	id + id) $	S5
0 T 2 * 7 (4 id 5	+ id) $	R6 (G[4,F])
0 T 2 * 7 (4 F 3	+ id) $	R4 (G[4,T])
0 T 2 * 7 (4 T 9	+ id) $	R1 (G[0,E])
0 T 2 * 7 (4 E 1	+ id) $	S6
0 T 2 * 7 (4 E 1 + 6	id) $	S5
0 T 2 * 7 (4 E 1 + 6 id 5	) $	R6 (G[6,F])
0 T 2 * 7 (4 E 1 + 6 F 3	) $	R4 (G[6,T])
0 T 2 * 7 (4 E 1 + 6 T 9	) $	R1 (G[4,E])
0 T 2 * 7 (4 E 8	) $	S11
0 T 2 * 7 (4 E 8) 11	$	R5 (G[7,F])
0 T 2 * 7 F 10	$	R3 (G[0,T])
0 T 2	$	R2 (G[0,E])
0 E 1	$	Accept

Build the LR Parsing Table

- Algorithm
  - Find the follow sets of all non-terminals.
  - Add a dummy rule $S \rightarrow A\$$, (A is the start symbol), and put a dot (.) in front of A.
  - Whenever the dot is in front of a non-terminal $A \rightarrow \alpha.B\beta$, add all rules of B with a dot $B \rightarrow .\gamma$ to the state.
  - Move the dot over one symbol X in the state s, $A \rightarrow \alpha.X\beta \Rightarrow A \rightarrow \alpha X.\beta$ and find the corresponding state s' (or make a new state if not exist).
    - If X is a terminal, add Shift s' action to Action(s,X).
    - If X is a non-terminal, add s' to Goto(s,X).
    - If the dot reaches the end of a rule $A \rightarrow \alpha$, put Reduce action to Action(s,a) for each element a in the follow of A.

LR Parsing Table

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id



	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				**			
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			

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

- Put a dot at the beginning of the start rule.
- Move the dot over the next symbol, and expand.

Summary

- Syntax analysis is a common part of language implementation
- A lexical analyzer is a pattern matcher that isolates small-scale parts of a program
  - Detects syntax errors
  - Produces a parse tree
- A recursive-descent parser is an LL parser
  - EBNF
- Parsing problem for bottom-up parsers: find the substring of current sentential form
- The LR family of shift-reduce parsers is the most common bottom-up parsing approach

Calculator LR Parser

Grammar

```
<statement> → <command> | <expr> | id | id ‘=‘ <expr>
<command> → ‘list’ | ‘clear’ | ‘exit’ | ‘quit’
<expr> → <expr> ‘+’ <term> | <expr> ‘-‘ <term> | <term>
<term> → <term> ‘*’ <factor> | <term> ‘/’ <factor> | <factor>
<factor> → <unary_op> <base> <exponent>
<unary_op> → ‘+’ | ‘-‘ | ε
<base> → ‘(‘ <expr> ‘)’ | id | number
<exponent> → ‘^’ <factor> | ε
```

Calculator LR Parser

Modified Grammar

```
<start> → <statement> $
<statement> → command
<statement> → id ‘=‘ <expr>
<statement> → <expr>
<expr> → <expr> ‘+’ <term>
<expr> → <expr> ‘-‘ <term>
<expr> → <term>
<term> → <term> ‘*’ <factor>
<term> → <term> ‘/’ <factor>
<term> → <factor>
<factor> → <base>
<factor> → <base> ‘^’ <factor>
<factor> → ‘+’ <base>
<factor> → ‘+’ <base> ‘^’ <factor>
<factor> → ‘-’ <base>
<factor> → ‘-’ <base> ‘^’ <factor>
<base> → id
<base> → number
<base> → ‘(‘ <expr> ‘)’
```

Modified Grammar

```
 1. A → S $
 2. S → cmd
 3. S → id = E
 4. S → E
 5. E → E + T
 6. E → E - T
 7. E → T
 8. T → T * F
 9. T → T / F
10. T → F
11. F → B
12. F → B ^ F
13. F → + B
14. F → + B ^ F
15. F → - B
16. F → - B ^ F
17. B → id
18. B → num
19. B → ( E )
```

Modified Grammar

A → S $

1. S → cmd
2. S → id = E
3. S → E
4. E → E + T
5. E → E - T
6. E → T
7. T → T * F
8. T → T / F
9. T → F
10. F → B
11. F → B ^ F
12. F → + B
13. F → + B ^ F
14. F → - B
15. F → - B ^ F
16. B → id
17. B → num
18. B → (E)

[0]

A → . S $	G1
S → . cmd	S2
S → . id = E	S3
S → . E	G4
E → . E + T	G4
E → . E – T	G4
E → . T	G5
T → . T * F	G5
T → . T / F	G5
T → . F	G6
F → . B	G7
F → . B ^ F	G7
F → . + B	S8
F → . + B ^ F	S8
F → . – B	S9
F → . - B ^ F	S9
B → . id	S3
B → . num	S10
B → . (E)	S11

[1]

A → S . $	**

[2]

S → cmd .	R1

[3]

S → id . = E	S12
B → id .	R16

[4]

S → E .	R3
E → E . + T	S13
E → E . - T	S14

[5]

E → T .	R6
T → T . * F	S15
T → T . / F	S16

[6]

T → F .	R9

[7]

F → B .	R10
F → B . ^ F	S17

[8]

F → + . B	G18
F → + . B ^ F	G18
B → . id	S20
B → . num	S10
B → . (E)	S11

[9]

F → - . B	G19
F → - . B ^ F	G19
B → . id	S20
B → . num	S10
B → . (E)	S11

[10]

B → num .	R17

[11]

B → (. E)	G21
E → . E + T	G21
E → . E - T	G21
E → . T	G5
T → . T * F	G5
T → . T / F	G5
T → . F	G6
F → . B	G7
F → . B ^ F	G7
F → . + B	S8
F → . + B ^ F	S8
F → . - B	S9
F → . - B ^ F	S9
B → . id	S20
B → . num	S10
B → . (E)	S11

[12]

S → id = . E	G22
E → . E + T	G22
E → . E - T	G22
E → . T	G5
T → . T * F	G5
T → . T / F	G5
T → . F	G6
F → . B	G7
F → . B ^ F	G7
F → . + B	S8
F → . + B ^ F	S8
F → . - B	S9
F → . - B ^ F	S9
B → . id	S20
B → . num	S10
B → . (E)	S11

[13]

E → E + . T	G23
T → . T * F	G23
T → . T / F	G23
T → . F	G6
F → . B	G7
F → . B ^ F	G7
F → . + B	S8
F → . + B ^ F	S8
F → . - B	S9
F → . - B ^ F	S9
B → . id	S20
B → . num	S10
B → . (E)	S11

[14]

E → E - . T	G24
T → . T * F	G5
T → . T / F	G5
T → . F	G6
F → . B	G7
F → . B ^ F	G7
F → . + B	S8
F → . + B ^ F	S8
F → . - B	S9
F → . - B ^ F	S9
B → . id	S20
B → . num	S10
B → . (E)	S11

[15]

T → T * . F	G25
F → . B	G7
F → . B ^ F	G7
F → . + B	S8
F → . + B ^ F	S8
F → . - B	S9
F → . - B ^ F	S9
B → . id	S20
B → . num	S10
B → . (E)	S11

[16]

T → T / . F	G26
F → . B	G7
F → . B ^ F	G7
F → . + B	S8
F → . + B ^ F	S8
F → . - B	S9
F → . - B ^ F	S9
B → . id	S20
B → . num	S10
B → . (E)	S11

[17]

F → B ^ . F	G27
F → . B	G7
F → . B ^ F	G7
F → . + B	S8
F → . + B ^ F	S8
F → . - B	S9
F → . - B ^ F	S9
B → . id	S20
B → . num	S10
B → . (E)	S11

[18]

F → + B .	R12
F → + B . ^ F	S28

[19]

F → - B .	R14
F → - B . ^ F	R29

[20]

B → id .	R16

[21]

B → (E .)	S30
E → E . + T	S13
E → E . - T	S14

[22]

S → id = E .	R2
E → E . + T	S13
E → E . - T	S14

[23]

E → E + T .	R4
T → T . * F	S15
T → T . / F	S16

[24]

E → E – T .	R5
T → T . * F	S15
T → T . / F	S16

[25]

T → T * F .	R7

[26]

T → T / F .	R8

[27]

F → B ^ F .	R15

[28]

F → + B ^ . F	G31
F → . B	G7
F → . B ^ F	G7
F → . + B	S8
F → . + B ^ F	S8
F → . - B	S9
F → . - B ^ F	S9
B → . id	S20
B → . num	S10
B → . (E)	S11

[29]

F → - B ^ . F	G32
F → . B	G7
F → . B ^ F	G7
F → . + B	S8
F → . + B ^ F	S8
F → . - B	S9
F → . - B ^ F	S9
B → . id	S20
B → . num	S10
B → . (E)	S11

[30]

B → (E) .	R18

[31]

F → + B ^ F .	R13

[32]

F → - B ^ F .	R15

$\text{FOLLOW}(S) = \{\ \$\ \}$
$\text{FOLLOW}(E) = \{\ \$, +, -,)\ \}$
$\text{FOLLOW}(T) = \{\ \$, +, -, *, /,)\ \}$
$\text{FOLLOW}(F) = \{\ \$, +, -, *, /,)\ \}$
$\text{FOLLOW}(B) = \{\ \$, +, -, *, /, \hat{},)\ \}$

	Action												Goto				
	id	num	cmd	=	+	-	*	/	^	(	)	$	S	E	T	F	B
0	S3	S10	S2		S8	S9							1	4	5	6	7
1												**					
2												R1					
3				S12	R16	R16	R16	R16	R16		R16	R16					
4					S13	S14						R3					
5					R6	R6	S15	S16			R6	R6					
6					R9	R9	R9	R9			R9	R9					
7					R10	R10	R10	R10	S17		R10	R10					
8	S20	S10								S11							18
9	S20	S10								S11							19
10					R17	R17	R17	R17	R17		R17	R17					
11	S20	S10			S8	S9				S11				21	5	6	7
12	S20	S10			S8	S9				S11				22	5	6	7
13	S20	S10			S8	S9				S11					23	6	7
14	S20	S10			S8	S9				S11					24	6	7
15	S20	S10			S8	S9				S11						25	7
16	S20	S10			S8	S9				S11						26	7
17	S20	S10			S8	S9				S11						27	7
18					R12	R12	R12	R12	S28		R12	R12					
19					R14	R14	R14	R14	S29		R14	R14					
20					R16	R16	R16	R16	R16		R16	R16					
21					S13	S14					S30						
22					S13	S14						R2					
23					R4	R4	S15	S16			R4	R4					
24					R5	R5	S15	S16			R5	R5					
25					R7	R7	R7	R7			R7	R7					
26					R8	R8	R8	R8			R8	R8					
27					R15	R15	R15	R15			R15	R15					
28	S20	S10			S8	S9				S11						31	7
29	S20	S10			S8	S9				S11						32	7
30					R18	R18	R18	R18	R18		R18	R18					
31					R13	R13	R13	R13			R13	R13					
32					R15	R15	R15	R15			R15	R15					