



ĐẠI HỌC BÁCH KHOA HÀ NỘI  
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

# Unit 9.

## Recursive descent parsing

# Characteristics

- Used to parse LL(1) language
- Can be extended for parsing LL(k) grammars, but algorithms are complicated
- Parsing non LL(k) grammars can cause infinite loops

# Recursive-descent parsing

- A top-down parsing method
- The term *descent* refers to the direction in which the parse tree is traversed (or built).
- Use a set of *mutually recursive* procedures (one procedure for each nonterminal symbol)
  - Start the parsing process by calling the procedure that corresponds to the start symbol
  - Each production becomes one branch in procedure for its LHS
- We consider a special type of recursive-descent parsing called predictive parsing
  - Use a lookahead symbol to decide which production to use

# Recursive Descent Parsing

- For every BNF rule (production) of the form

$$\langle \text{phrase1} \rangle \rightarrow E$$

the parser defines a function to parse phrase1 whose body is to parse the rule E

```
void compilePhrase1( )  
{ /* parse the rule E */ }
```

- Where E consists of a sequence of non-terminal and terminal symbols
- Requires **no left recursion** in the grammar.

# Parsing a rule

- A sequence of non-terminal and terminal symbols,  
 $Y_1 Y_2 Y_3 \dots Y_n$   
is recognized by parsing each symbol in turn
- For each non-terminal symbol,  $Y$ , call the corresponding parse function `compileY`
- For each terminal symbol,  $y$ , call a function  
`eat(y)`  
that will check if  $y$  is the next symbol in the source program
  - The terminal symbols are the token types from the lexical analyzer
  - If the variable `currentsymbol` always contains the next token:
    - `eat(y):`
      - if (`currentsymbol == y`)
      - then `getNextToken()`
      - else `SyntaxError()`

# Simple parse function example

- Suppose that there was a grammar rule  
**Prog ::= KW PROGRAM Ident SB\_SEMICOLON  
Block SB\_PERIOD**
- Then the function for par

```
void compileProgram(void)  
{  
    eat(KW_PROGRAM) ;  
    eat(TK_IDENT) ;  
    eat(SB_SEMICOLON) ;  
    compileBlock() ;  
    eat(SB_PERIOD) ;  
}
```

# Look-Ahead

- In general, one non-terminal may have more than one production, so more than one function should be written to parse that non-terminal.
- Instead, we insist that we can decide which rule to parse just by looking ahead one symbol in the input

**BasicType ::= KW\_INTEGER | KW\_CHAR**

Then compileBasicType can have the form

```
switch (lookAhead->tokenType) {  
  case KW_INTEGER:  
    eat(KW_INTEGER);  
    break;  
  case KW_CHAR:  
    eat(KW_CHAR);  
    break;  
  default:  
    error(ERR_INVALIDBASICTYPE, lookAhead->lineNo, lookAhead->colNo);  
    break;  
}
```

# KPL Parser

- Can be built using BNF rules or syntax diagrams
- Use syntax diagrams: consists of 13 functions, each function for a syntax diagram
- Use BNF rules: consist of approximate 50 functions, each function for a variable (non-terminal symbol)



# Compile factor function

//Follow rules of slide 19 unit 8

```
void compileFactor(void) {  
    switch (lookAhead->tokenType) {  
        case TK_NUMBER:  
            eat(TK_NUMBER);  
            break;  
        case TK_CHAR:  
            eat(TK_CHAR);  
            break;  
        case TK_IDENT:  
            eat(TK_IDENT);  
            switch (lookAhead->tokenType) {  
                case SB_LSEL:  
                    compileIndexes();  
                    break;  
                case SB_LPAR:  
                    compileArguments();  
                    break;  
                default: break;  
            }  
        break;  
    }
```

```
        case SB_LPAR:  
            eat(SB_LPAR);  
            compileExpression();  
            eat(SB_RPAR);  
            break;  
        default:  
            error(ERR_INVALIDFACTOR, lookAhead->lineNo, lookAhead->colNo);  
    }  
}
```

# BNF Rules for term

82)  $\text{Term} ::= \text{Factor Term2}$

83)  $\text{Term2} ::= \text{SB\_TIMES Factor Term2}$

84)  $\text{Term2} ::= \text{SB\_SLASH Factor Term2}$

85)  $\text{Term2} ::= \varepsilon$

# compileTerm and compileTerm2 functions

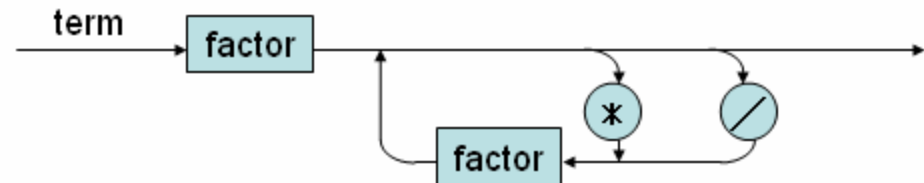
```
void compileTerm(void) {
    compileFactor();
    compileTerm2();
}

void compileTerm2(void) {
    switch (lookAhead->tokenType) {
        case SB_TIMES:
            eat(SB_TIMES);
            compileFactor();
            compileTerm2();
            break;
        case SB_SLASH:
            eat(SB_SLASH);
            compileFactor();
            compileTerm2();
            break;
```

```
// check the FOLLOW set
        case SB_PLUS:
        case SB_MINUS:
        case KW_TO:
        case KW_DO:
        case SB_RPAR:
        case SB_COMMA:
        case SB_EQ:
        case SB_NEQ:
        case SB_LE:
        case SB_LT:
        case SB_GE:
        case SB_GT:
        case SB_RSEL:
        case SB_SEMICOLON:
        case KW_END:
        case KW_ELSE:
        case KW_THEN:
            break;
        default:
            error(ERR_INVALIDTERM, lookAhead->lineNo,
                lookAhead->coIno);
    }
}
```

# CompileTerm function (Use SD)

```
void compileTerm(void)
{
    compileFactor();
    while(lookAhead->tokenType== SB_TIMES ||
        lookAhead->tokenType == SB_SLASH)
    {switch (lookAhead->tokenType)
    {
        case SB_TIMES:
            eat(SB_TIMES);
            compileFactor();
            break;
        case SB_SLASH:
            eat(SB_SLASH);
            break;
    }
}
```



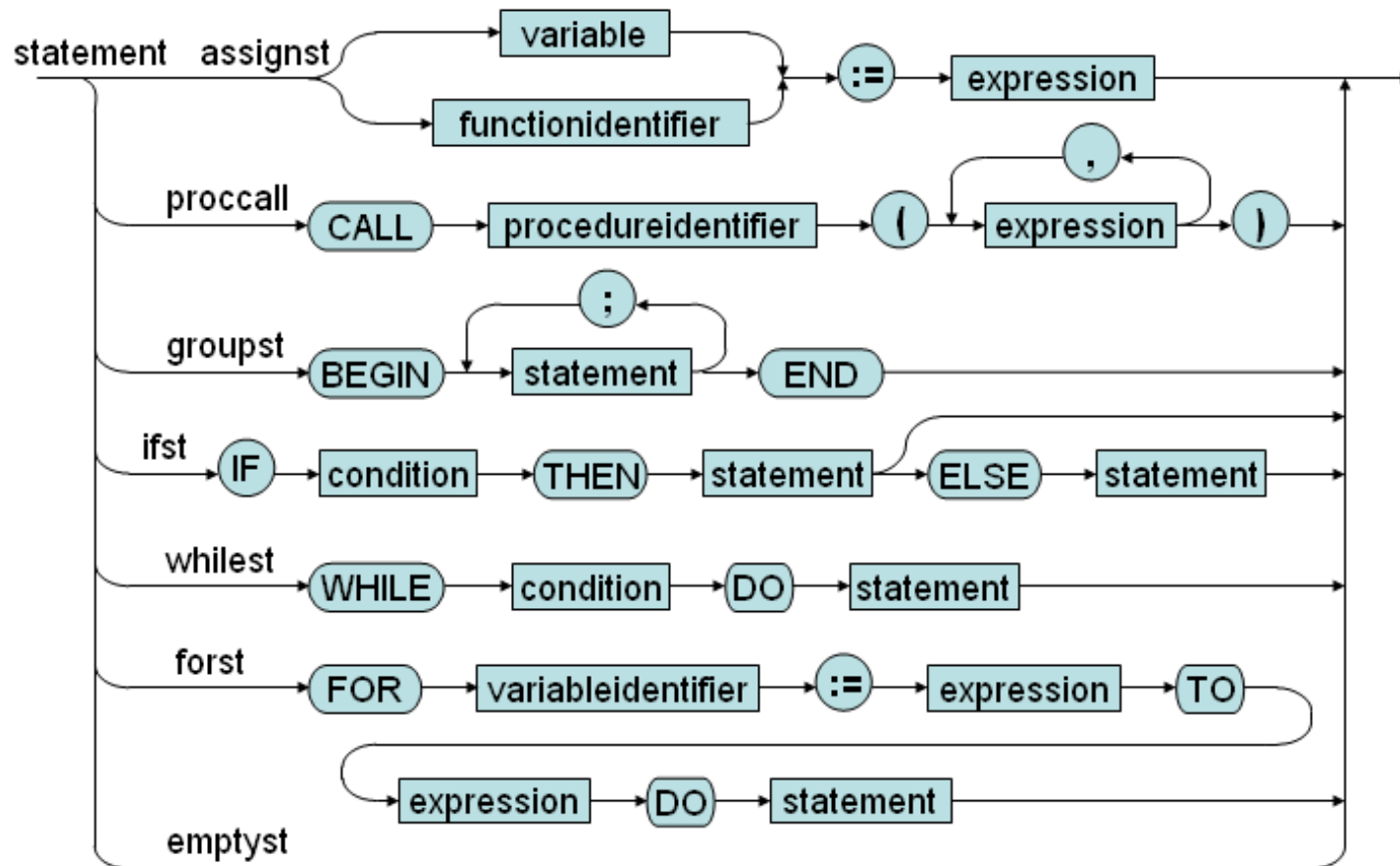
# Statement

- 49) Statement ::= AssignSt
- 50) Statement ::= CallSt
- 51) Statement ::= GroupSt
- 52) Statement ::= IfSt
- 53) Statement ::= WhileSt
- 54) Statement ::= ForSt
- 55) Statement ::=  $\epsilon$

# compileStatement function (using BNF)

```
void compileStatement(void) {  
    switch (lookAhead->tokenType) {  
        case TK_IDENT:  
            compileAssignSt();  
            break;  
        case KW_CALL:  
            compileCallSt();  
            break;  
        case KW_BEGIN:  
            compileGroupSt();  
            break;  
        case KW_IF:  
            compileIfSt();  
            break;  
        case KW_WHILE:  
            compileWhileSt();  
            break;  
        case KW_FOR:  
            compileForSt();  
            break;  
        // EmptySt needs to check FOLLOW tokens  
        case SB_SEMICOLON:  
            case KW_END:  
            case KW_ELSE:  
                break;  
                // Error occurs  
            default:  
                error(ERR_INVALIDSTATEMENT, lookAhead->lineNo, lookAhead->colNo);  
                break;  
    }  
}
```

# Syntax diagram for statement



# compileStatement function (using SD)

```
void compileStatement(void) {
    switch (lookAhead->tokenType) {
case TK_IDENT:
    eat(TK_IDENT);
    while (lookAhead->tokenType==SB_LSEL)
        {eat(SB_LSEL);
        compileExpression();
        eat(SB_RSEL); }
    eat(SB_ASSIGN);
    compileExpression();
    break;
case KW_CALL:
    eat(KW_CALL);
    eat(TK_IDENT);
    if (lookAhead->tokenType== SB_LPAR)
        eat(SB_LPAR);
        compileExpression();
        while (lookAhead->tokenType== SB_COMMA)
            {eat (SB_COMMA);
            compileExpression();}
        eat(SB_RPAR);
        // Check FOLLOW set .....
    break;
case KW_BEGIN:.....
    break;
case KW_IF:.....
    break;
case KW_WHILE:.....
    break;
case KW_FOR:.....
    break;
// EmptySt needs to check FOLLOW tokens
case SB_SEMICOLON:
case KW_END:
case KW_ELSE:
    break;
    // Error occurs
default:
    error(ERR_INVALIDSTATEMENT, lookAhead->lineNo,
lookAhead->coIno);
    break;
    }
}
```

