Bài 9.
Phương pháp đệ quy trên xuống

Đặc điểm của phương pháp

- Sử dụng để phân tích cú pháp cho các văn phạm LL(1)
- Có thể mở rộng cho văn phạm LL(k), nhưng việc tính toán phức tạp
- Sử dụng để phân tích văn phạm khác có thể dẫn đến lặp vô hạn

Bộ phân tích cú pháp

- Bao gồm một tập thủ tục, mỗi thủ tục ứng với một sơ đồ cú pháp (một ký hiệu không kết thúc)
- Các thủ tục đệ quy : khi triển khai một ký hiệu không kết thúc có thể gặp các ký hiệu không kết thúc khác, dẫn đến các thủ tục gọi lẫn nhau, và có thể gọi trực tiếp hoặc gián tiếp đến chính nó.

Mô tả chức năng

- Giả sử mỗi thủ tục hướng tới một đích ứng với một ký hiệu không kết thúc
- Tại mỗi thời điểm luôn có một đích được triển khai, kiểm tra cú pháp hết một đoạn nào đó trong văn bản nguồn(vẽ được cây phân tích cú pháp đến một mức nào đó

Thủ tục triển khai một đích

- Đối chiếu văn bản nguồn với một đường trên sơ đồ cú pháp
- Đọc từ tố tiếp
- Đối chiếu với nút tiếp theo trên sơ đồ
  - Nếu là nút tròn (ký hiệu kết thúc)thì từ tố vừa đọc phải phù hợp với từ tố trong nút
  - Nếu là nút chữ nhật nhãn A (ký hiệu không kết thúc), từ tố vừa đọc phải thuộc FIRST (A) => tiếp tục triển khai đích A
- Ngược lại, thông báo một lỗi cú pháp tại điểm đang xét

Từ sơ đồ thành thủ tục

- Mỗi sơ đồ ứng với một thủ tục
- Các nút xuất hiện tuần tự chuyển thành các câu lệnh kế tiếp nhau.
- Các điểm rẽ nhánh chuyển thành câu lệnh lựa chọn (if, case)
- Chu trình chuyển thành câu lệnh lặp (while, do while, repeat. . .)
- Nút tròn chuyển thành đoạn đối chiếu từ tố
- Nút chữ nhật chuyển thành lời gọi tới thủ tục khác

Chú ý

- Bộ phân tích cú pháp luôn đọc trước một từ tố
- Xem trước một từ tố cho phép chọn đúng đường đi khi gặp điểm rẽ nhánh trên sơ đồ cú pháp
- Khi thoát khỏi thủ tục triển khai một đích, có một từ tố đã được đọc dôi ra
- Hàm đối chiếu từ tố: eat-> kiểm tra xem kiểu của từ tố đọc trước có phù hợp với kiểu của từ tố được sinh bởi luật không. Nếu có, đọc từ tố tiếp, ngược lại : báo lỗi

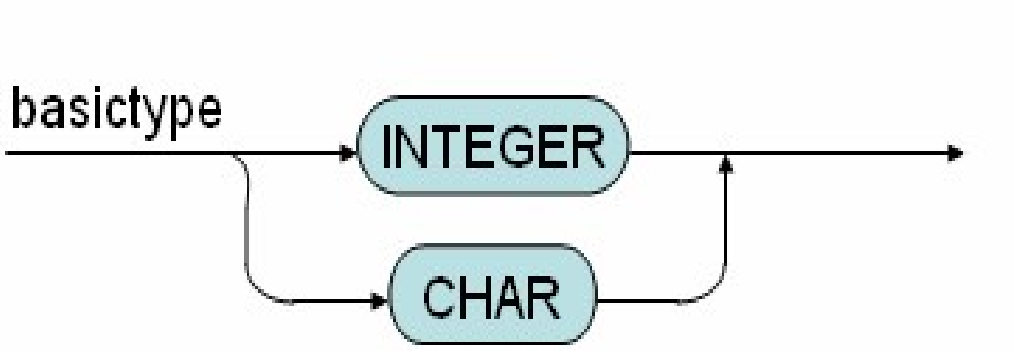
Hàm eat - duyệt ký hiệu kết thúc

```
void eat(TokenType tokenType) {
  if (lookAhead->tokenType == tokenType) {
    printToken(lookAhead);
    scan();
  } else missingToken(tokenType, lookAhead->lineNo,
lookAhead->colNo);
}
```

Thủ tục CompileBasicType

```
34) BasicType ::= KW_INTEGER
35) BasicType ::= KW_CHAR
```

```
void compileBasicType
{switch(lookahead->tokenType )
{case KW_INTEGER:
eat(KW_INTEGER);
case KW_CHAR:
eat(KW_CHAR);bre
default: error()
}}
```



Cú pháp statement

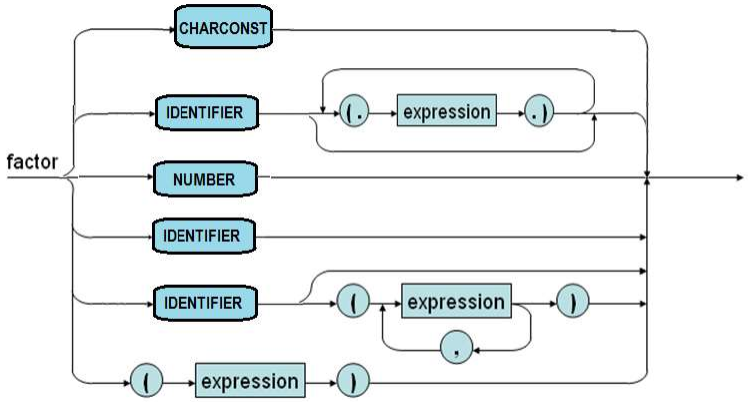
```
49) Statement ::= AssignSt
50) Statement ::= CallSt
51) Statement ::= GroupSt
52) Statement ::= IfSt
53) Statement ::= WhileSt
54) Statement ::= ForSt
55) Statement ::= ε
```

Phân tích statement

```
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;
    // check FOLLOW tokens
  case SB_SEMICOLON:
  case KW_END:
  case KW_ELSE:
    break;
    // Error occurs
  default:
    error(ERR_INVALIDSTATEMENT,
lookAhead->lineNo, lookAhead-
>colNo);
    break;
  }
}
```

Hàm compileFactor

```
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);
  }
}
```

Hàm compileTerm (Sử dụng BNF)

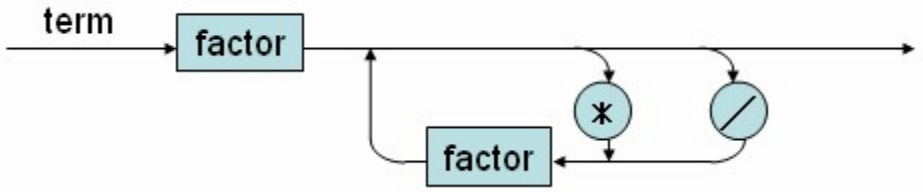
```
82) Term ::= Factor Term2

83) Term2 ::= SB_TIMES Factor Term2
84) Term2 ::= SB_SLASH Factor Term2
85) Term2 ::= ε
```

Các hàm compileTerm và CompileTerm2 (dùng BNF

```
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->colNo);
  }
}
```

Hàm CompileTerm (dùng sơ đồ cú pháp)

```
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);
    compileFactor();
    break;}}}
```



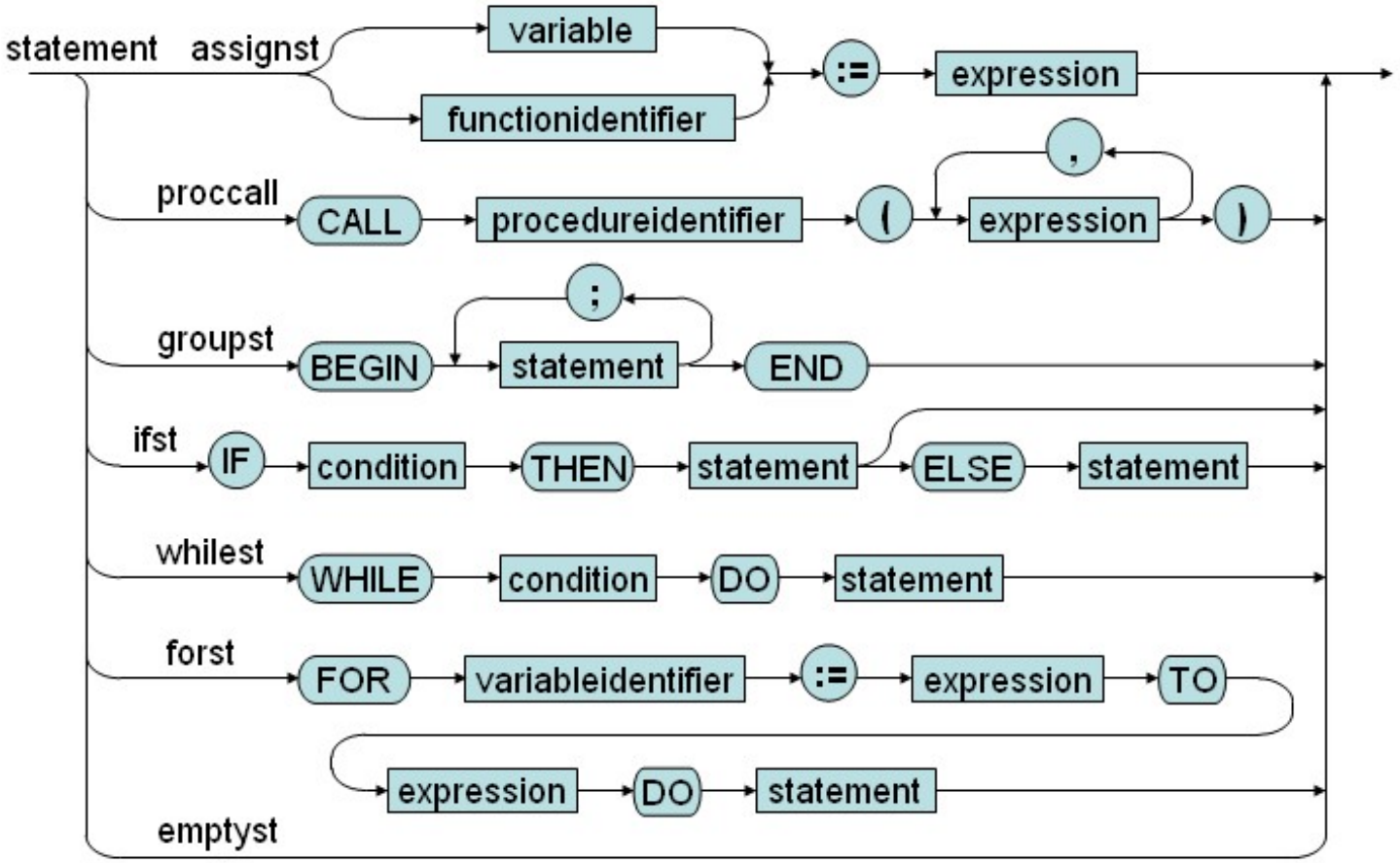
Các luật cú pháp của lệnh

```
49) Statement ::= AssignSt
50) Statement ::= CallSt
51) Statement ::= GroupSt
52) Statement ::= IfSt
53) Statement ::= WhileSt
54) Statement ::= ForSt
55) Statement ::= ε
```

Hàm compileStatement (dùng 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;
   // Các xử lý cho FOLLOW
   case SB_SEMICOLON:
   case KW_END:
   case KW_ELSE:
     break;
     // Báo lỗi
   default:
     error(ERR_INVALIDSTATEMENT,
lookAhead->lineNo, lookAhead-
>colNo);
     break;
  }
}
```

Sơ đồ cú pháp cho lệnh



Hàm compileStatement (Dùng SĐCP)

```
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)
      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->colNo);
    break;
  }
}
```