


# 프로그래밍언어론

## Chapter 3 어휘와 구문분석

*Syntactic sugar causes cancer of the semicolon.*  
*A. Perlis*



## 목차

3.1 Chomsky Hierarchy

3.2 Lexical Analysis

3.3 Syntactic Analysis

## 구문 분석

- 파싱(파서)라고도 함.
- 목적: 원시 프로그램의 구조를 파악하는 것
- 입력: 토큰
- 출력: 파스 트리 또는 추상구문트리
- 재귀하향 파서는 문법의 각 논터미널에 해당하는 함수들로 구현되는데, 그 함수는 주어진 입력이 해당 논터미널로부터 생성가능한 지를 인식한다.

## 프로그램 구성

- 수식:  $x + 2 * y$
- 저장문:  $z = x + 2 * y$
- 반복문: `while (i < n) a[i++] = 0;`
- 함수 정의
- 선언문: `int i;`

## 저장문 문법

- *Assignment*  $\rightarrow$  *Identifier* = *Expression*
- *Expression*  $\rightarrow$  *Term* { *AddOp* *Term* }
- *AddOp*  $\rightarrow$  + | -
- *Term*  $\rightarrow$  *Factor* { *MulOp* *Factor* }
- *MulOp*  $\rightarrow$  \* | /
- *Factor*  $\rightarrow$  [ *UnaryOp* ] *Primary*
- *UnaryOp*  $\rightarrow$  - | !
- *Primary*  $\rightarrow$  *Identifier* | *Literal* | ( *Expression* )

## First 집합

- 확장 문법:  $S$  - 시작기호

$S' \Rightarrow S \$$

- $\text{First}(X)$ 는  $X$ 로부터 유도가능한 문장형태의 맨 왼쪽 단말자 집합이다.

$$\text{First}(X) = \{ a \in T \mid X \Rightarrow^* a w, w \in (N \cup T)^* \}$$

- 단말자 기호에 대한 생성규칙은 없기 때문에,

$$\text{First}(a) = \{ a \} \quad (a \in T)$$

## First 집합

- 단말자와 비단말자로 된 임의의 스트링  $w = X_1 \dots X_n V$  에 대하여
$$\text{First}(w) = \text{First}(X_1) \cup \dots \cup \text{First}(X_n) \cup \text{First}(V)$$
여기서  $X_1, \dots, X_n$ 은 널이 될 수 있고  $V$ 는 널이 될 수 없다.
- 널이 될 수 있다(nullable)는 것은 빈 스트링을 유도할 수 있음을 의미한다.

## Nullable Algorithm

- Set nullable = new Set( );
- do { oldSize = nullable.size( );
- for (Production p : grammar productions( ))
- if (p.length( ) == 0)
- nullable.add(p.nonterminal( ));
- else
- << check righthand side >>
- } while (nullable.size( ) > oldSize);



## 널이 될 수 있는 비단말자 집합 구하기

```
int oldSize;
Set nullable = new Set();
do {
    oldSize = nullable.size();
    for (Production p : grammar productions()) {
        boolean allNull = true;
        for (Symbol t : p.rule())
            if (! nullable.contains(t))
                allNull = false;
        if (allNull)
            nullable.add(p.nonterminal());
    }
} while (nullable.size() > oldSize); // nullable grew
```

## 문법 변환 규칙

- 확장 문법으로 수정
- 비단말자 이름을 간략화
- 메타 구문을 비단말자로 대체

## 변환된 문법(그림 3.8)

$S \rightarrow A \$$

$A \rightarrow i = E ;$

$E \rightarrow T E'$

$E' \rightarrow \mid AO T E'$

$AO \rightarrow + \mid -$

$T \rightarrow F T'$

$T' \rightarrow MO F T'$

$MO \rightarrow * \mid /$

$F \rightarrow F' P$

$F' \rightarrow \mid UO$

$UO \rightarrow - \mid !$

$P \rightarrow i \mid l \mid ( E )$

## 널이 될 수 있는 비단말자 집합 (그림 3.8에 대하여)

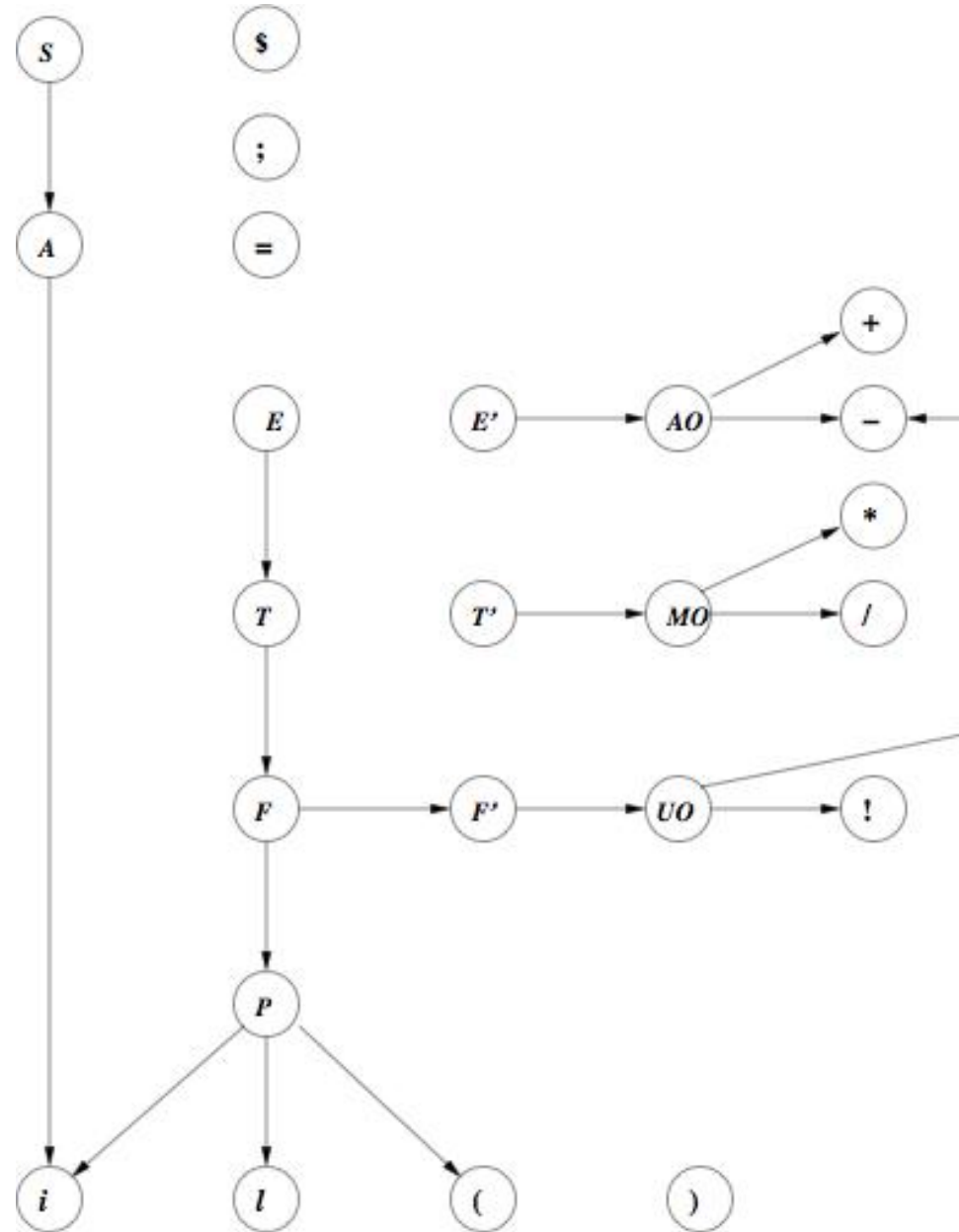
Pass	Nullable
------	----------

1	$E' T' F'$
---	------------

2	$E' T' F'$
---	------------

## 왼쪽 종속 그래프(Left Dependency Graph)

- 각 생성규칙  $A \rightarrow V_1 \dots V_n X w$  에 대하여
  - $V_1, \dots, V_n$ 이 널이 될 수 있으면  $A$ 에서  $X$ 로 가는 아크를 그린다.
  - $A$ 에서  $V_1, \dots, A$ 에서  $V_n$ 으로의 아크도 생성됨



**비단말자**

**First**

**비단말자**

**First**

A

i

UO

! -

E

! - i l (

P

i l (

E'

+ -

AO

+ -

T

! - i l (

T'

\* /

MO

\* /

F

! - i l (

F'

! -

## 재귀하향 파싱

- 각 비단말자에 대응하는 메소드/함수를 구현
- 비단말자로부터 유도가능한 가장 긴 토큰열을 인식
- 생성규칙을 코드로 변환하는 알고리즘이 필요
- EBNF을 기반



## 변환 알고리즘 $T(\text{EBNF}) = \text{Code}$

(생성규칙  $A \rightarrow w$ 에 대하여)

1.  $w$ 가 비단말자이면 이를 호출하라.
2.  $w$ 가 단말자이면, 주어진 토큰과 매치한다.
3.  $w$ 가  $\{w'\}$ 이면 다음 코드로 변환:  
    while (First( $w'$ )에 토큰이 있으면)  
    {  $T(w')$ ; }

## 변환 알고리즘 $T(\text{EBNF}) = \text{Code}$ (생성규칙 $A \rightarrow w$ 에 대하여)

4.  $w$  가  $w_1 \mid \dots \mid w_n$  이면 다음 코드로 변환,

```
switch (token) {  
    case First( $w_1$ ):  $T(w_1)$ ; break;  
    ...  
    case First( $w_n$ ):  $T(w_n)$ ; break;  
    default: error(token);  ( $w_i$ 가 빈 것이 있으면 default: break;)  
}
```

5.  $w$  가  $[w']$  이면  $(\mid w')$ 로 고치고 규칙 4를 적용.

6.  $w = X_1 \dots X_n$  이면,  $T(w) = T(X_1); \dots T(X_n)$ ; 를 적용

## 확장 생성규칙의 변환

- 첫번째 토큰을 구한다.
- 본래의 시작기호에 해당하는 메소드를 호출한다.
- 마지막 토큰이 파일끝 토큰인지 검사한다.

## 메소드 match

```
private void match (int t) {  
    if (token.type() == t)  
        token = lexer.next();  
    else  
        error(t);  
}
```

## 메소드 error

```
private void error(int tok) {  
    System.err.println(  
        "Syntax error: expecting"  
        + tok + "; saw: " + token);  
    System.exit(1);  
}
```

## 배정문 파서

```
private void assignment( ) {  
    // Assignment □ Identifier = Expression ;  
    match(Token.Identifier);  
    match(Token.Assign);  
    expression( );  
    match(Token.Semicolon);  
}
```

## 수식 파서

```
private void expression( ) {  
    // Expression  $\rightarrow$  Term { AddOp Term }  
  
    term( );  
    while (isAddOp()) {  
        token = lexer.next( );  
        term( );  
    }  
}
```

## 구문구조와 의미구조의 연결

- 파스트리 는 비효율적
- 우선순위 단계마다 하나의 비단말자가 존재
- 파스트리의 모양이 중요



## 파스트리 간단히 하기(추상구문 만들기)

- 분리자/구두기호 제거
- 불필요한 루트 비단말자 제거
- 남은 비단말자를 단말자로 대체

## 추상구문

*Assignment* = *Variable* target; *Expression* source

*Expression* = *Variable* | *Value* | *Binary* | *Unary*

*Binary* = *Operator* op; *Expression* term1, term2

*Unary* = *Operator* op; *Expression* term

*Variable* = *String* id

*Value* = *Integer* value


*Operator* = + | - | \* | / | !



```
abstract class Expression { }
```

```
class Binary extends Expression {  
    Operator op;  
    Expression term1, term2;  
}
```


```
class Unary extends Expression {  
    Operator op; Expression term;  
}
```




## AST를 출력하는 수정판 T


(생성규칙  $A \rightarrow w$  에 대하여)

1. 함수 이름을 A라고 하고 A에 대응하는 추상구문 너미널을 이 함수의 반환타입으로 한다.
2. w 가 비단말자이면 반환된 값을 저장.
3. w 가 구두기호가 아닌 즉, 추상구문에도 나오는 단말자이면 그 값을 저장.
4. 적절한 추상구문트리를 생성하도록 객체를 생성하여 반환하는 문장을 포함시킨다.



```
private Assignment assignment( ) {  
    // Assignment  $\square$  Identifier = Expression ;  
    Variable target = match(Token.Identifier);  
    match(Token.Assign);  
    Expression source = expression( );  
    match(Token.Semicolon);  
    return new Assignment(target, source);  
}
```





```
private String match (int t) {  
    String value = Token.value( );  
    if (token.type( ) == t)  
        token = lexer.next( );  
    else  
        error(t);  
    return value;  
}
```

