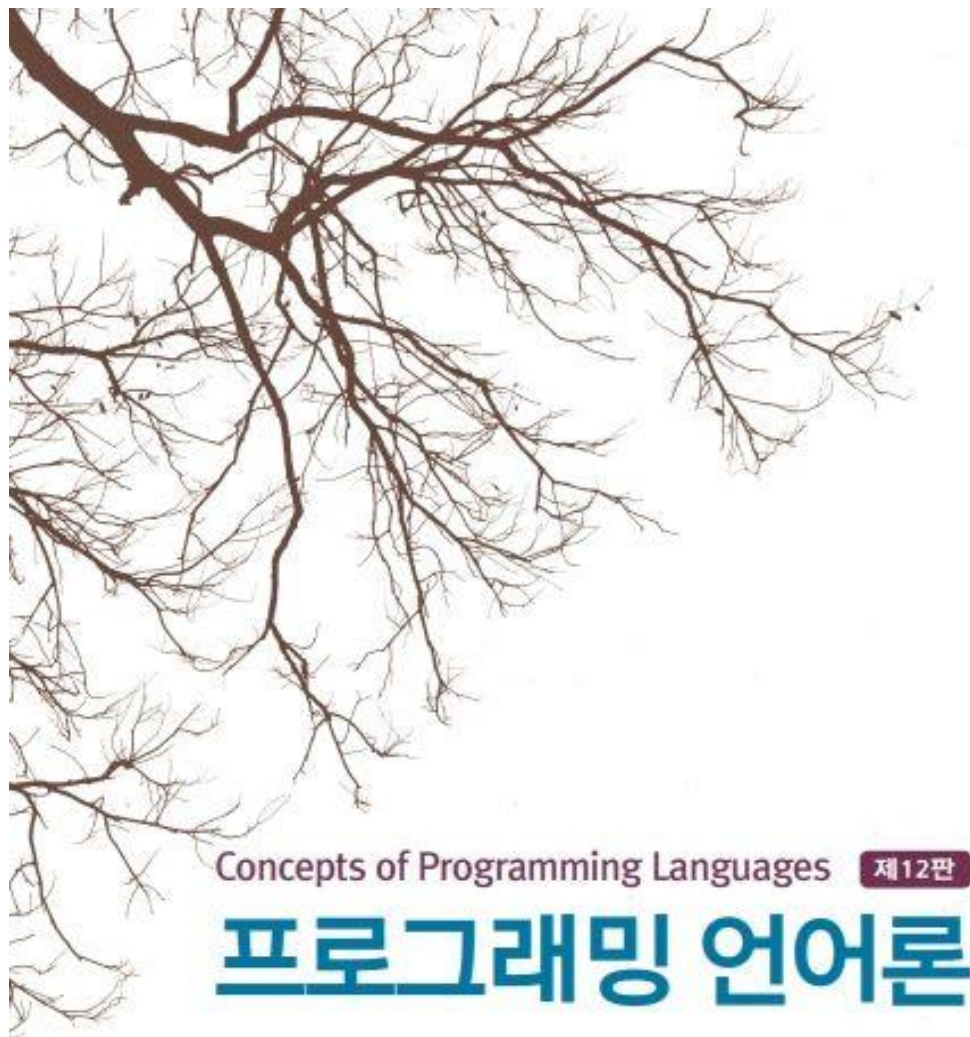


3장

구문론

4장

어휘분석과
구문분석



Robert W. Sebesta 지음 | 하상호 · 김진성 · 송하윤 · 우균 · 이덕우 · 조장우 옮김

구문 및 구문법

QnA

- 구문법(Syntax)
 - 문장 혹은 프로그램을 작성하는 방법
 - 자연어(영어, 한국어)의 문법처럼 프로그래밍 언어의 구문법이 있다.
 - 프로그래밍 언어의 이론적 기초
- 질문
 - 어떤 언어의 가능한 문장 혹은 프로그램의 개수가 무한하지 않나요?
 - 무한한 것들을 어떻게 유한하게 정의할 수 있나요?

재귀적 정의: 이진수의 구문법

- 숫자(D)는 0, 1 중 하나이다.
- 이진수 구성 방법
 - (1) 숫자(D)는 이진수(N)이다.
 - (2) 이진수(N) 다음에 숫자(D)가 오면 이진수(N)이다.
- 논리 규칙 형태

$$\frac{D \text{는 숫자이다}}{D \text{는 이진수 } N \text{이다}}$$

$$\frac{N \text{이 이진수이고 } D \text{가 숫자이다}}{ND \text{는 이진수이다}}$$

- 문법 형태

$N \rightarrow D$
 $N \rightarrow ND$



$N \rightarrow D$
 $\mid ND$

이진수: 구문법과 의미론

이진수 구문법

$D \rightarrow 0$
| 1

$N \rightarrow D$
| ND

● 101

이진수의 의미 : 십진수 값

$$V('0') = 0$$

$$V('1') = 1$$

$$V(D)$$

$$V(ND) = V(N) * 2 + V(D)$$

$$V('101') = V('10') * 2 + V('1') = 2 * 2 + 1 = 5$$

$$V('10') = V('1') * 2 + V(0) = 2$$

십진수: 구문법과 의미론

$D \rightarrow 0$

| 1

| 2

...

| 9

$N \rightarrow D$

| ND

$V('0') = 0$

$V('1') = 1$

$V('2') = 2$

...

$V('9') = 9$

$V(D)$

$V(ND) = V(N) * 10 + V(D)$

● 386

$V('386') = 386$

수식의 구문법

- 좀 현실적인 구문은 없나요?
- 수식
 - 5, 5 + 13, 5 + 13 + 4, 5 * 13 + 4, (5 + 13), (5 + 13) * 12, ...
- 구문법 : 쓰는 방법

$$\begin{array}{l} E \rightarrow E * E \\ \quad | \quad E + E \\ \quad | \quad (E) \\ \quad | \quad N \end{array}$$
$$N \rightarrow N D \mid D$$
$$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

수식의 의미

수식의 구문법

$E \rightarrow E * E$
| $E + E$
| (E)
| N

수식의 의미(시맨틱스)

$V(E * E) = V(E) * V(E)$
 $V(E + E) = V(E) + V(E)$
 $V((E)) = V(E)$
 $V(N)$

- 수식의 의미는 무엇일까요?

$$\begin{aligned} V('3 * 5 + 12') &= V('3 * 5') + V('12') = V('3') * V('5') + V('12') \\ &= 3 * 5 + 12 = 27 \end{aligned}$$

프로그래밍 언어의 구문구조

- 프로그래밍 언어의 구문 구조를 어떻게 표현할 수 있을까 ?
 - 재귀를 이용한 구문법으로 정의
- 문장 S 의 구문법
 - $id = E$
 - if E then S else S
 - while E do S
- 문맥-자유 문법(CFG: Context-free grammar)
 - 이러한 재귀 구조를 자연스럽게 표현할 수 있다.

$S \rightarrow id = E$
| if E then S else S
| while (E) S

문맥-자유 문법 CFG

- 문맥-자유 문법 CFG는 다음과 같이 구성된다.

- 터미널 심볼의 집합 T
- 논터미널 심볼의 집합 N
- 시작 심볼 S (논터미널 심볼 중에 하나)
- 다음과 같은 형태의 생성(문법) 규칙들의 집합

$X \rightarrow Y_1 Y_2 \dots Y_n$ 여기서 $X \in N$ 그리고 $Y_i \in T \cup N$

$X \rightarrow \epsilon$ (오른쪽이 빈 스트링인 경우)

- 보통 논터미널(nonterminal) 심볼은 대문자로,
터미널(terminal) 심볼은 소문자로 표기한다.

생성 규칙

- 생성 규칙(production rule) 또는 문법 규칙
 - $X \rightarrow Y_1 Y_2 \dots Y_n$
 - X 를 작성하는 방법을 정의하는 문법 규칙
 - X 는 $Y_1 Y_2 \dots Y_n$ 형태로 작성할 수 있다는 것을 의미한다.
- 문장 S

$S \rightarrow \text{id} = E$	$S \rightarrow \text{id} = E$
$S \rightarrow \text{if } E \text{ then } S \text{ else } S$	$\text{if } E \text{ then } S \text{ else } S$
$S \rightarrow \text{while } E \text{ do } S$	$\text{while } E \text{ do } S$

[언어 S의 문장 요약 문법 1]



Stmt $S \rightarrow id = E$

| $S; S$

| $\text{if } E \text{ then } S$

| $\text{if } E \text{ then } S \text{ else } S$

| $\text{while } E \text{ do } S$

| $\text{read } id$

| $\text{print } E$

Expr $E \rightarrow n \mid id \mid \text{true} \mid \text{false}$

| $E + E \mid E - E \mid E * E \mid E / E \mid (E)$

| $E == E \mid E != E \mid E < E \mid E > E \mid !E$

유도

유도(Derivation)

- 입력된 문장 혹은 프로그램이 문법에 맞는지 검사하는 것을 구문검사라고 한다.
- **QnA** 어떤 문장 혹은 프로그램이 구문법에 맞는지는 어떻게 검사할 수 있을까?
- 입력된 스트링이 문법에 맞는지 검사하려면 문법으로부터 **유도 (derivation)**해 보아야 한다.
- **[핵심 개념]** 어떤 스트링이 문법으로부터 유도 가능하면 문법에 맞는 스트링이고 그렇지 않으면 문법에 맞지 않는 스트링이다.

유도(Derivation)

- 핵심 아이디어
 1. 시작 심볼 S 부터 시작한다.
 2. 논터미널 심볼 X 를 생성규칙을 적용하여 $Y_1 Y_2 \dots Y_n$ 으로 대체한다.
 3. 이 과정을 논터미널 심볼이 없을 때까지 반복한다.
- 생성 규칙 $X \rightarrow Y_1 Y_2 \dots Y_n$ 적용
 - X 를 $Y_1 Y_2 \dots Y_n$ 으로 대체한다. 혹은
 - X 가 $Y_1 Y_2 \dots Y_n$ 을 생성한다.
- 터미널 심볼
 - 대체할 규칙이 없으므로 일단 생성되면 끝
 - 터미널 심볼은 그 언어의 토큰이다.
- 예
 - $S \rightarrow aS \mid b$
 - $S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaab$

유도(Derivation)

- 직접 유도(Direct derivation) \Rightarrow

- 생성 규칙을 한 번 적용
- 생성 규칙 $X_i \rightarrow Y_1 Y_2 \dots Y_n$ 이 존재하면

$$X_1 \dots X_i \dots X_n \Rightarrow X_1 \dots X_{i-1} Y_1 Y_2 \dots Y_n X_{i+1} \dots X_n$$

- 유도(Derivation) \Rightarrow^*

- 생성 규칙을 여러 번 적용
- $X_1 \dots X_n \Rightarrow \dots \Rightarrow Y_1 \dots Y_m$ 이 가능하면 $X_1 \dots X_n \Rightarrow^* Y_1 \dots Y_m$

유도 예제

- CFG

$$E \rightarrow E * E \quad (1)$$

$$| E + E \quad (2)$$

$$| (E) \quad (3)$$

$$| N \quad (4)$$

$$N \rightarrow N D \mid D$$

$$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

- 생성할 스트링: $3 + 4 * 5$

- 유도

$$E \Rightarrow E + E \Rightarrow N + E \Rightarrow D + E \Rightarrow 3 + E \Rightarrow 3 + E * E \Rightarrow \dots \Rightarrow 3 + 4 * 5$$

- $3 + 4 + 5$ 유도?

좌측 유도과 우측 유도

- 좌측 유도(leftmost derivation)

- 각 직접 유도 단계에서 가장 왼쪽 언터미널을 선택하여 이를 대상으로 생성 규칙을 적용한다.

- 우측 유도(rightmost derivation)

- 각 직접 유도 단계에서 가장 오른쪽 언터미널을 선택하여 이를 대상으로 생성 규칙을 적용하면 된다.

- $3 + 4 * 5$ 의 우측 유도

- $$\begin{aligned} E &\Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * N \Rightarrow {}^* E + E * 5 \\ &\Rightarrow E + N * 5 \Rightarrow {}^* E + 4 * 5 \Rightarrow N + 4 * 5 \Rightarrow {}^* 3 + 4 * 5 \end{aligned}$$

문법 G의 언어

- 문법 G에 의해서 정의되는 언어 $L(G)$
 - 문법 G에 의해서 유도되는 모든 스트링들의 집합
 - $L(G) = \{a_1 \dots a_n \mid S \Rightarrow^* a_1 \dots a_n, \text{ 모든 } a_i \text{ 는 터미널 심볼이다.}\}$
 - 예: 문법 G
 - $S \rightarrow (S)$
 - $S \rightarrow a$
- (1) 먼저 몇 개의 가능한 스트링을 유도(생성)해 보면 다음과 같다.
- $S \Rightarrow a$
 - $S \Rightarrow (S) \Rightarrow (a)$
 - $S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow ((a))$
 - ...
- (2) 이들을 집합 형태로 표현해 보자.
- $L(G) = \{a, (a), ((a)), (((a))), \dots\} = \{ (^n a)^n \mid n \geq 0 \}$

유도 트리

- 유도 트리(Derivation tree)
 - 유도 과정 혹은 구문 구조를 보여주는 트리
 - 유도 트리 = 파스 트리 = 구문 트리
- 유도는 시작 심볼로부터 시작하여 연속적으로 직접 유도를 한다.

$$S \Rightarrow \dots \Rightarrow \dots$$

- 이러한 유도 과정은 다음과 같이 트리 형태로 그릴 수 있다.
 - (1) S 가 트리의 루트이다.
 - (2) 규칙 $X \rightarrow Y_1 Y_2 \dots Y_n$ 을 적용하여 직접 유도를 할 때마다 X 노드는 Y_1, \dots, Y_n 를 자식 노드로 갖도록 트리를 구성한다.

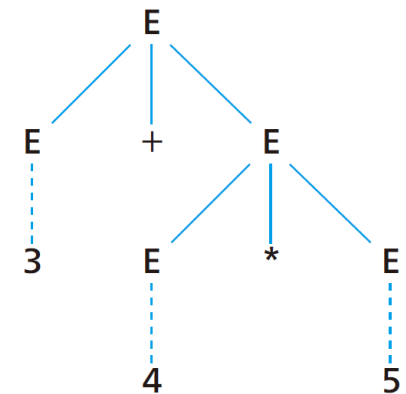
유도 트리 예제

- CFG

$$\begin{aligned} E &\rightarrow E * E \\ &\quad | \quad E + E \\ &\quad | \quad (E) \\ &\quad | \quad N \end{aligned}$$
$$N \rightarrow N D \mid D$$
$$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

- 생성할 스트링: $3 + 4 * 5$

- 유도

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow N + E \Rightarrow^* 3 + E \\ &\Rightarrow 3 + E * E \Rightarrow^* 3 + 4 * 5 \end{aligned}$$


$3 + 4 * 5$ 를 위한 파스 트리

유도 트리에 대한 참조

- 이 트리 구조는 $3 + (4 * 5)$ 와 같은 결합 성질을 보여준다.
- 주의
 - 좌측 유도와 우측 유도 모두 같은 파스트리를 갖는다.
 - 차이점은 파스트리에 가지가 추가되는 순서이다.

모호성

모호성(Ambiguity)

- 수식을 위한 문법

$$\begin{array}{l} E \rightarrow E * E \\ | E + E \\ | (E) \\ | N \end{array}$$

- 예

$$3 + 4 * 5$$

- 이 스트링은 **두 개의 좌측 유도**를 갖는다.

$$(1) E \Rightarrow E + E \Rightarrow N + E \Rightarrow^* 3 + E \Rightarrow 3 + E * E \Rightarrow^* 3 + 4 * 5$$

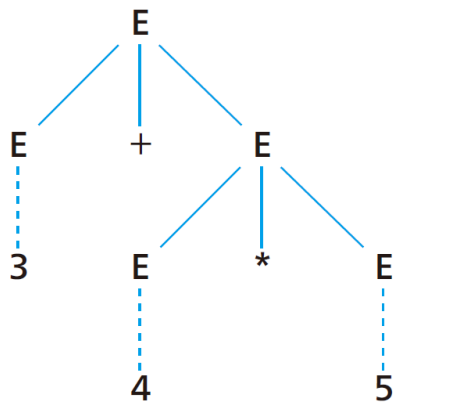
$$(2) E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow N + E * E \Rightarrow 3 + E * E \Rightarrow^* 3 + 4 * 5$$

- 이 스트링은 **두 개의 파스트리**를 갖는다.

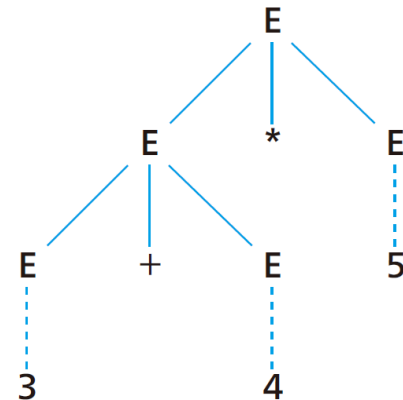
모호성(Ambiguity)

- 모호한 문법(ambiguous grammar)

- 어떤 스트링에 대해 **두 개 이상의 좌측 유도**를 갖는다.
- 어떤 스트링에 대해 **두 개 이상의 우측 유도**를 갖는다.
- 어떤 스트링에 대해 **두 개 이상의 파스 트리**를 갖는다.



3 + 4 * 5의 첫 번째 파스 트리



3 + 4 * 5의 두 번째 파스 트리

- 모호성은 나쁘다

- 왜 ?

모호성 처리 방법 1

- 문법 재작성

- 원래 언어와 같은 언어를 정의하면서 모호하지 않도록 문법 재작성

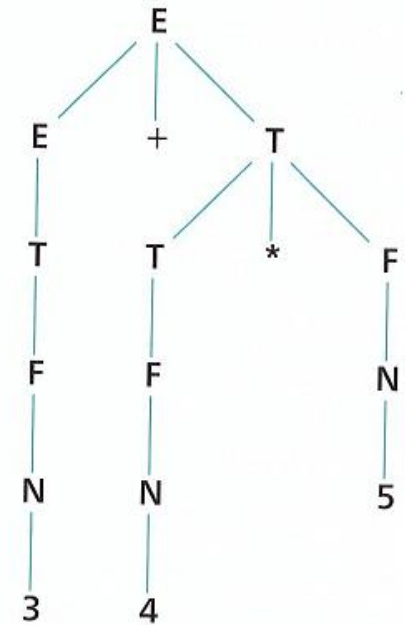
- 예

- 우선 순위를 적용하여 모호하지 않도록 재작성
- 수식은 여러 개의 항들을 더하는 구조이다.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow N \mid (E)$$



3 + 4 * 5에 대한 유도 트리

- 3 + 4 * 5의 좌측 유도

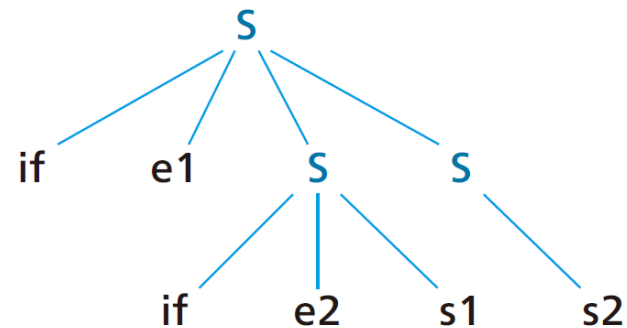
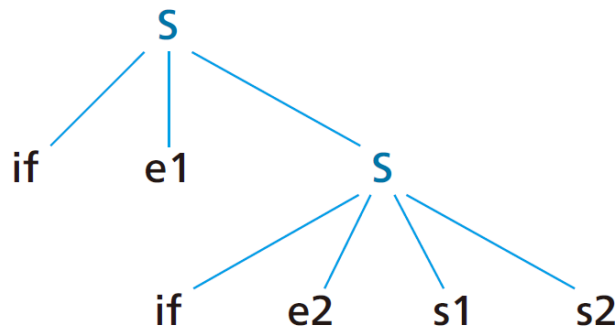
- $E \Rightarrow E + T \Rightarrow^* N + T \Rightarrow 3 + T \Rightarrow 3 + T * F \Rightarrow 3 + F * F$
 $\Rightarrow 3 + N * F \Rightarrow^* 3 + 4 * N \Rightarrow^* 3 + 4 * 5$

모호성 예: The Dangling Else

- 모호한 문법

$S \rightarrow$ if E then S
| if E then S else S

- 이 문장에 대한 두 개의 파스 트리
if e1 then if e2 then s1 else s2



if 문: 모호한 문법

모호성 처리 방법 2

- 언어 구문 일부 변경
 - 원래 언어와 약간 다른 언어를 정의하도록
 - 언어의 구문을 일부 변경하여
 - 모호하지 않은 문법 작성

$S \rightarrow \text{if } E \text{ then } S \text{ end}$
| $\text{if } E \text{ then } S \text{ else } S$

- 작성 예
 - $\text{if } e1 \text{ then if } e2 \text{ then } s1 \text{ else } s2 \text{ end}$
 - $\text{if } e1 \text{ then if } e2 \text{ then } s1 \text{ end else } s2$

BNF와 구문 다이어그램

BNF/EBNF

- BNF(Backus–Naur Form)

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$

$\langle \text{factor} \rangle \rightarrow \text{number} \mid (\langle \text{expr} \rangle)$

- EBNF(Extended BNF)

$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \{ + \langle \text{term} \rangle \}$

$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ * \langle \text{factor} \rangle \}$

$\langle \text{factor} \rangle \rightarrow \text{number} \mid (\langle \text{expr} \rangle)$

- [핵심 개념]

[] : 0번 혹은 1번 (optional)

{ } : 0번 이상 반복

[언어 S 문법 2:EBNF]

$\langle \text{stmt} \rangle \rightarrow \text{id} = \langle \text{expr} \rangle ;$
| ' { ' $\langle \text{stmt} \rangle$ ' }'
| if ($\langle \text{expr} \rangle$) then $\langle \text{stmt} \rangle$ [else $\langle \text{stmt} \rangle$]
| while ($\langle \text{expr} \rangle$) $\langle \text{stmt} \rangle$
| read id;
| print $\langle \text{expr} \rangle ;$

$\langle \text{expr} \rangle \rightarrow \langle \text{bexp} \rangle \{ \& \langle \text{bexp} \rangle \mid ' \mid ' \langle \text{bexp} \rangle \} \mid ! \langle \text{expr} \rangle \mid \text{true} \mid \text{false}$
 $\langle \text{bexp} \rangle \rightarrow \langle \text{aexp} \rangle [\langle \text{relop} \rangle \langle \text{aexp} \rangle]$
 $\langle \text{relop} \rangle \rightarrow == \mid != \mid < \mid > \mid <= \mid >=$
 $\langle \text{aexp} \rangle \rightarrow \langle \text{term} \rangle \{ + \langle \text{term} \rangle \mid - \langle \text{term} \rangle \}$
 $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ * \langle \text{factor} \rangle \mid / \langle \text{factor} \rangle \}$
 $\langle \text{factor} \rangle \rightarrow [-] (\text{number} \mid ' (' \langle \text{aexp} \rangle ') ' \mid \text{id})$

구문 다이어그램

- 구문 다이어그램

- 각 생성규칙을 다이어그램으로 표현
- 논터미널 => 사각형
- 터미널 => 원
- 순서 => 화살표

- 수식 문법 EBNF

$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \{ + \langle \text{term} \rangle \mid - \langle \text{term} \rangle \}$

$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ * \langle \text{factor} \rangle \mid / \langle \text{factor} \rangle \}$

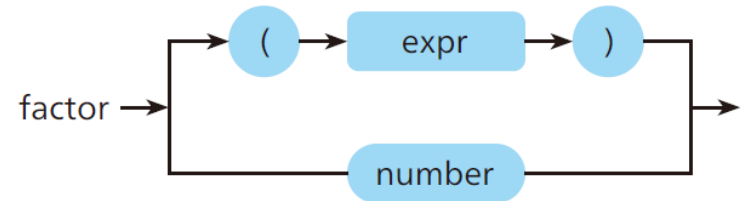
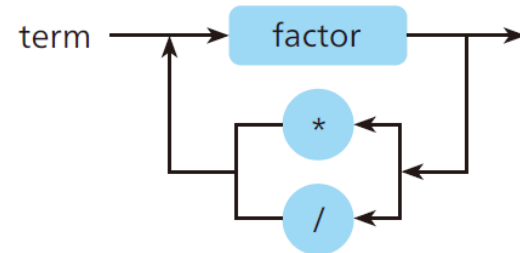
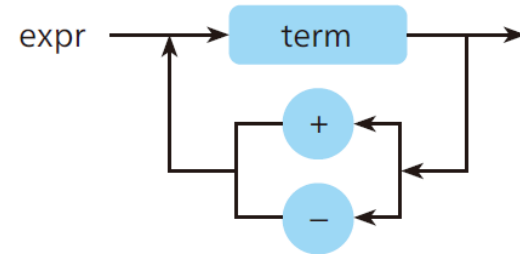
$\langle \text{factor} \rangle \rightarrow \text{number} \mid (\langle \text{expr} \rangle)$

구문 다이어그램

- 수식 문법 EBNF

$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \{ + \langle \text{term} \rangle \mid - \langle \text{term} \rangle \}$
 $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ * \langle \text{factor} \rangle \mid / \langle \text{factor} \rangle \}$
 $\langle \text{factor} \rangle \rightarrow \text{number} \mid (\langle \text{expr} \rangle)$

- EBNF에서 중괄호로 나타낸 반복
- 다이어그램에서는 루프를 사용
- expr를 위한 다이어그램
 - 화살표를 따라가면서 루프를 돌아
 - term을 여러 번 반복할 수 있다.



수식 문법의 구문 다이어그램

재귀 하강 파싱

지금까지 한 것/앞으로 할 것!

● 주제

논리

구현

● 구문법

문법

파서

● 의미론

의미 함수

인터프리터

● 타입

타입 규칙

타입 검사기

재귀 하강 파싱(recursive-descent parsing)

- 파싱
 - 입력 스트링을 유도하여 문법에 맞는지 검사
- 파서
 - 입력 스트링을 유도하여 문법에 맞는지 검사하는 프로그램
- 재귀 하강 파서의 기본 원리
 - 입력 스트링을 좌측 유도(leftmost derivation)하도록 문법으로부터 직접 파서 프로그램을 만든다.

재귀 하강 파싱 구현

- 각 년터미널
 - 하나의 프로시저(함수, 메소드)를 구현한다.
- 프로시저 내에서
 - 생성규칙 우변을 적용하여 좌우선 유도 하도록 작성한다.

$\langle A \rangle \rightarrow \langle B \rangle \ c \ \langle D \rangle$ \longrightarrow

```
A( )  
{  
    B( );  
    match("c");  
    D( );  
}
```

- 프로시저 호출
 - 생성 규칙을 적용하여 유도
- match(문자);
 - 다음 입력(토큰)이 문자와 일치하는지 검사

예제

- 수식을 재귀-하강 파싱
- `<command> → <expr> '\n'`

```
void command(void)
{
    int result = expr( );
    if (token == '\n')
        printf("The result is: %d\n", result);
    else error();
}
```

```
void parse(void)
{
    token = getToken();
    command();
}
```

```
main()
{
    parse();
    return 0;
}
```

예제

- $\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \{ + \langle \text{term} \rangle \}$

```
void expr(void)
```

```
{
```

```
    term( );
```

```
    while (token == '+') {
```

```
        match('+');
```

```
        term();
```

```
    }
```

```
}
```

```
void match(int c)
```

```
{ // 현재 토큰 확인 후 다음 토큰 읽기
```

```
    if (token == c)
```

```
        token = getToken();
```

```
    else error();
```

```
}
```

어휘분석기 getToken()

```
int getToken() { // 다음 토큰(수 혹은 문자)을 읽어서 리턴한다.
    while(true) {
        try {
            ch = input.read();
            if (ch == ' ' || ch == '\t' || ch == '\r') ;
            else if (Character.isDigit(ch)) {
                value = number();
                input.unread(ch);
                return NUMBER;
            }
            else return ch;
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```


수식 값 계산기

- 수식 값 계산
 - 재귀-하강 파싱 하면서 동시에 수식의 값을 계산
- $\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \{ + \langle \text{term} \rangle \}$

```
int expr(void)
{
    int result = term( );
    while (token == '+') {
        match('+');
        result += term();
    }
    return result;
}
```

수식 값 계산기

- 항의 값 계산
 - 재귀-하강 파싱 하면서 동시에 항(term)의 값을 계산
- $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ * \langle \text{factor} \rangle \}$

```
int term(void)
{
    int result = factor( );
    while (token == '*') {
        match('*');
        result *= factor();
    }
    return result;
}
```

수식 값 계산기

- 인수 값 계산
 - 수 혹은 괄호 수식의 값 계산
 - $\langle \text{factor} \rangle \rightarrow \langle \text{number} \rangle \mid (\langle \text{expr} \rangle)$
 - $\langle \text{number} \rangle \rightarrow \langle \text{digit} \rangle \{ \langle \text{digit} \rangle \}$

- 사용 예

```
>> 12+33
```

```
45
```

```
>> 3*5+10
```

```
25
```

```
>> (2+3)*12
```

```
60
```

```
>> 2+3*12
```

```
38
```