제2장 구문법(Syntax)

숙명여대 창병모

2.1 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?

    ⇒ recursion

재귀적 정의: 이진수의 구문법

- 숫자(D)는 0, 1중 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

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

- 문법 형태

$$N \rightarrow D \quad N \rightarrow ND \quad \Longleftrightarrow \quad N \rightarrow D \mid ND$$

이진수: 구문법과 의미론

이진수 구문법

Syntax

① D → 0
| 1

② N → D
| ND

- 101
①②②

이진수의 의미 : 십진수 값

semantics을정의하는함수

(value)

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

Syntax

Semantics

① $D \rightarrow 0$
| 1
| 2
...
| 9

$V('0') = 0$
$V('1') = 1$
$V('2') = 2$
...
$V('9') = 9$

② $N \rightarrow D$
| ND

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

- 386
①②②

$V('386') = 386$

수식의 구문법

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

expression

E → E * E
  | E + E
  | (E)
  | N

N → N D | D

D → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

수식의 의미

수식의 구문법

$$
\begin{aligned}
E \rightarrow\ & E * E \\
\mid\ & E + E \\
\mid\ & (E) \\
\mid\ & N
\end{aligned}
$$

수식의 의미(시맨틱스)

semantics를 정의하는 함수

$$
\begin{aligned}
V(E * E) &= V(E) * V(E) \\
V(E + E) &= V(E) + V(E) \\
V((\ E\ )) &= V(E) \\
V(N)
\end{aligned}
$$

- 수식의 의미는 무엇일까요?

  V('3 * 5 + 12') = V('3 * 5') + V('12') = V('3') * V('5') + V('12')

  = 3 * 5 + 12 = 27

프로그래밍 언어의 구문구조

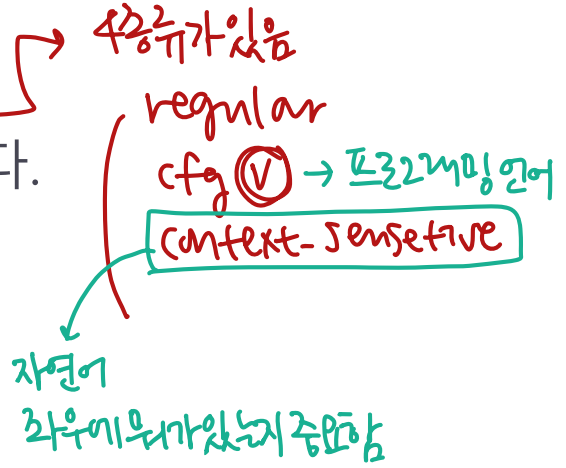
- 프로그래밍 언어의 구문 구조를 어떻게 표현할 수 있을까 ?
  - 재귀를 이용한 구문법으로 정의

identifier statement

- 문장 S의 구문법 : S를정의하는데에 S 사용(recursion)
  - id = E → assignment statement
  - 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 → 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*** : 문법규칙의 왼쪽에 올수없음
  - 넌터미널 심볼의 집합 ***N*** : 문법규칙의 왼쪽(해당하는 문법규칙 작성법이있음)
  - 시작 심볼 *S* (넌터미널 심볼 중에 하나)
  - 다음과 같은 형태의 생성(문법) 규칙들의 집합

    $X \rightarrow Y1\ Y2 \ldots Yn$ 여기서 $X \in N$ 그리고 $Yi \in T \cup N$

    $X \rightarrow \varepsilon$ (오른쪽이 빈 스트링인 경우) : empty rule
- 보통 넌터미널(nonterminal) 심볼은 대문자로,

  터미널(terminal) 심볼은 소문자로 표기한다.

생성 규칙 = 문법규칙

왜 생성규칙이라고 하는지?! → 생성문법에서 시작됐기때문

- 생성 규칙(production rule) 또는 문법 규칙
  - $X \rightarrow Y_1 Y_2 \ldots Y_n$
  - X를 작성하는 방법을 정의하는 문법 규칙
  - X는 $Y_1 Y_2 \ldots Y_n$ 형태로 작성할 수 있다는 것을 의미한다.

- 문장 S

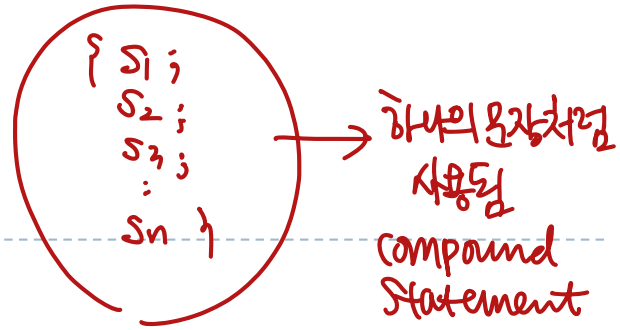
  S → id = E

  S → if E then S else S

  S → while E do S

  S → id = E
  | if E then S else S
  | while E do S

[언어 S의 문장 요약 문법 1]



문장(Statement)

Stmt S → id = E

| S; S → 세미콜론으로 불려져 있는 Statement

| if E then S

| if E then S else S

| while E do S

| read id 입력

| print E 출력

expression

Expr E → n | id | true | false

산술 | E + E | E - E | E * E | E / E | (E)

관계 | E == E | E != E | E < E | E > E | !E

2.2 유도

유도

- 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.

  문법규칙을 적용해 해당하는 String을 생성해본다

- **[핵심 개념]** 어떤 스트링이 문법으로부터 유도 가능하면 문법에 맞는 스트링이고 그렇지 않으면 문법에 맞지 않는 스트링이다.

  ⇒ 생성된 String은 모두 문법규칙에 맞음

유도(Derivation)

- 핵심 아이디어
  1. 시작 심볼 S부터 시작한다.
  2. 넌터미널 심볼 X를 생성규칙을 적용하여 $Y_1\ Y_2 \ldots\ Y_n$으로 대치한다.
  3. 이 과정을 넌터미널 심볼이 없을 때까지 반복한다.

- 생성 규칙 $X \rightarrow Y_1\ Y_2 \ldots\ Y_n$ 적용
  - X를 $Y_1\ Y_2 \ldots\ Y_n$으로 대치한다. 혹은
  - X가 $Y_1\ Y_2 \ldots\ Y_n$을 생성한다

⇒ 문법규칙이 생성규칙인 이유!!

- 터미널 심볼
  - 대치할 규칙이 없으므로 일단 생성되면 끝
  - 터미널 심볼은 그 언어의 토큰이다.

- 예
  - S → aS | b
  - S => aS => aaS => aaaS => aaab

문법규칙에 맞는걸까? 생성해보자.

ba : 안됨!

b, ab, aab, aaab, … ⇒ $\{a^n b \mid n \geq 0\}$

유도(Derivation)

- 직접 유도(Direct derivation) $\Rightarrow$
  - 생성 규칙을 한 번 적용
  - *생성규칙* $X_i \rightarrow Y_1\ Y_2 \ldots Y_n$*이 존재하면*

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

- 유도(Derivation) $\Rightarrow^*$ **: 0번이상*
  - 생성 규칙을 여러 번 적용
  - $X_1 \ldots X_n \Rightarrow \ldots \Rightarrow Y_1 \ldots Y_m$ 이 가능하면 $X_1 \ldots X_n \Rightarrow^* Y_1 \ldots Y_m$

유도 예제

- CFG

  E → E * E (1)
  | E + E (2)
  | (E) (3)
  | N (4)
  N → N D | D
  D → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- 생성할 스트링: 3 + 4 * 5
- 유도

  E ⇒ E + E ⇒ N + E ⇒ D + E ⇒ 3 + E ⇒ 3 + E * E ⇒ ... ⇒ 3 + 4 * 5

  (2) (4) E→N→D의 5개 (1)

- 3 + 4 + 5 유도 ?

  E ⇒ E+E ⇒ N+E ⇒ D+E ⇒ 3+E ⇒ 3+E+E ⇒ 3+E+N ⇒ 3+D+N ⇒ 3+4+E ⇒ 3+4+N ⇒ 3+4+D ⇒ 3+4+5

좌측 유도와 우측 유도

- 좌측 유도(leftmost derivation)
  - 각 직접 유도 단계에서 가장 왼쪽 넌터미널을 선택하여 이를 대상으로 생성 규칙을 적용한다.

- 우측 유도(rightmost derivation)
  - 각 직접 유도 단계에서 가장 오른쪽 넌터미널을 선택하여 이를 대상으로 생성 규칙을 적용하면 된다.

- 3 + 4 * 5의 우측 유도
  - $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * N \Rightarrow^{*} E + E * 5$
    $\Rightarrow E + N * 5 \Rightarrow^{*} E + 4 * 5 \Rightarrow N + 4 * 5 \Rightarrow^{*} 3 + 4 * 5$

문법 G 언어

- 문법 G에 의해서 정의되는 언어 L(G)
  - 문법 G에 의해서 유도되는 모든 스트링들의 집합 (터미널스트링)
  - L(G) = {*a*1 … *an* | S ⇒* *a*1 … *an*, *모든 ai* 는 터미널 심볼이다.}

- 예: 문법 G
  - S → (S)
  - S → a

(1) 먼저 몇 개의 가능한 스트링을 유도(생성)해 보면 다음과 같다.

- S ⇒ a
- S ⇒ (S) ⇒ (a)
- S ⇒ (S) ⇒ ((S)) ⇒ ((a))
- …

(2) 이들을 집합 형태로 표현해 보자.

- L(G) = {a, (a), ((a)), (((a))), … } = { $({}^{n}\,a\,)^{n}$ | n≥0 }

유도 트리

derivation → 구현하면 parser

- 유도 트리(Derivation tree)
  - 유도 과정 혹은 구문 구조를 보여주는 트리
  - 유도 트리 = 파스 트리 = 구문 트리

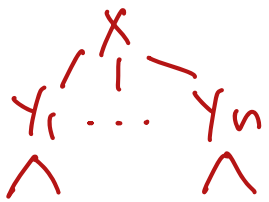
- 유도는 시작 심볼로부터 시작하여 연속적으로 직접 유도를 한다.

$$S \Rightarrow \ldots \Rightarrow \ldots$$

- 이러한 유도 과정은 다음과 같이 트리 형태로 그릴 수 있다.

  (1) S가 트리의 루트이다.

  (2) 규칙 $X \rightarrow Y_1\ Y_2 \ldots Y_n$을 적용하여 직접 유도를 할 때마다
  X 노드는 $Y_1, \ldots, Y_n$를 자식 노드로 갖도록 트리를 구성한다.



유도 트리 예제

- CFG

  E → E * E
  | E + E
  | (E)
  | N

  N → N D | D

  D → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- 생성할 스트링: 3 + 4 * 5
- 유도

  E ⇒ E + E ⇒ N + E ⇒* 3 + E
  ⇒ 3 + E * E ⇒* 3 + 4 * 5

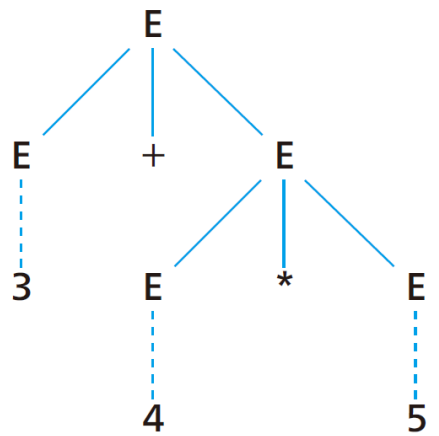


그림 2.1 3 + 4 * 5를 위한 파스 트리

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.

2.3 모호성

모호성(Ambiguity)

- 수식을 위한 문법

  E → E * E
  | E + E
  | (E)
  | N

- 예

  3 + 4 * 5

  - 이 스트링은 두 개의 좌측 유도를 갖는다.

(1) E ⇒ E + E ⇒ N + E ⇒* 3 + E ⇒ 3 + E * E ⇒* 3 + 4 * 5

(2) E ⇒ E * E ⇒ E + E * E ⇒ N + E * E ⇒ 3 + E * E ⇒* 3 + 4 * 5

  - 이 스트링은 두 개의 파스트리를 갖는다.

모호성(Ambiguity)

- 모호한 문법(ambiguous grammar)
  - 어떤 스트링에 대해 두 개 이상의 좌측 유도를 갖는다.
  - 어떤 스트링에 대해 두 개 이상의 우측 유도를 갖는다.
  - 어떤 스트링에 대해 두 개 이상의 파스 트리를 갖는다.

일대일대응관계

√ 채택 : 연산자의 우선순위

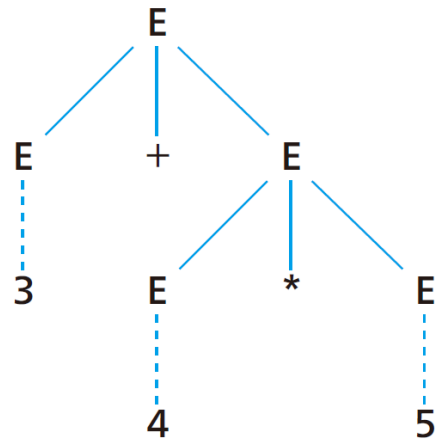


그림 2.2 3 + 4 * 5의 첫 번째 파스 트리

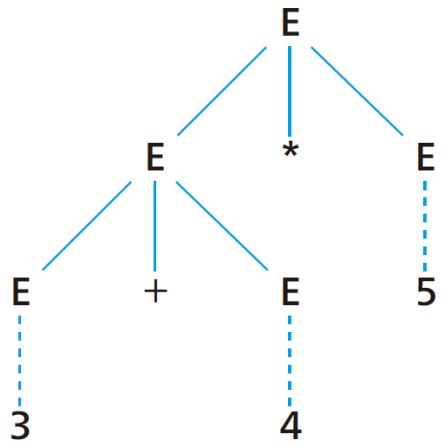


그림 2.3 3 + 4 * 5의 두 번째 파스 트리

- 모호성은 나쁘다
  - 왜 ? → 어떤 스트링에 대한 구문구조를 두개 이상으로 해석할 수 있음

↓ 모호하지 않도록 재작성/수정

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.

↳ term

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

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

인수 $F \rightarrow N \mid (E)$

number 괄호로묶인형태

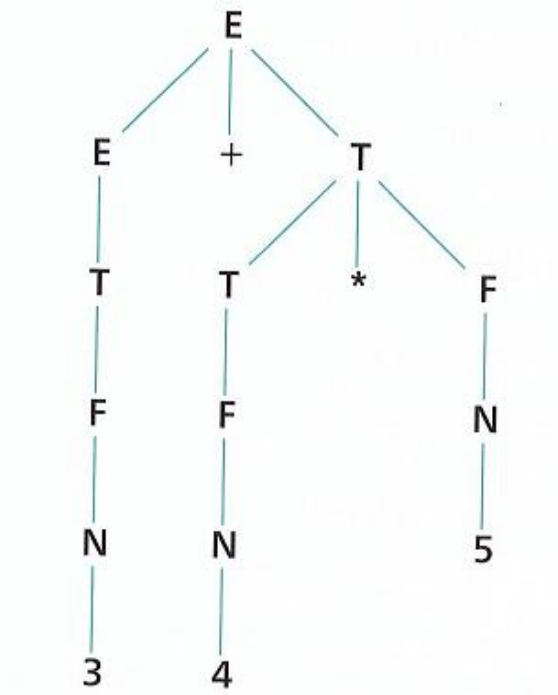


그림 2.4 3 + 4 * 5에 대한 유도 트리

- 3 + 4 * 5의 좌측 유도
  - $E => E + T =>^* N + T => 3 + T => 3 + T * F => 3 + F * F$
    $=> 3 + N * F =>^* 3 + 4 * N =>^* 3 + 4 * 5$

모호성 예: The Dangling Else

- 모호한 문법

S → 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

① ②

어떤 then과 매칭할까?

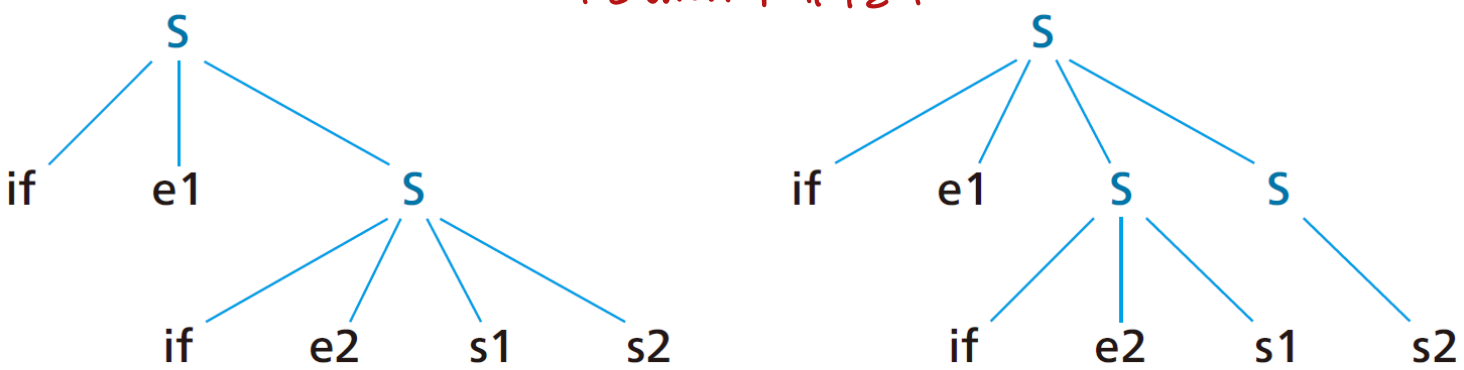


그림 2.5 if 문: 모호한 문법

모호성 처리 방법 2

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

S → if E then S end
  | if E then S else S

- 작성 예
  - if e1 then (if e2 then s1 else s2) end
  - if e1 then (if e2 then s1 end) else s2

연습문제 9 (교재 p747 풀어보기
pdf기준 p86

2.4 BNF와 구문 다이어그램

BNF/EBNF

- BNF(Backus-Naur Form) : 넌터미널을 <> 로 작성 → 긴 이름도 분명히 표현

재귀를 풀어서

<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → number | (<expr>)

- EBNF(Extended BNF) : {...} 는 중괄호 안의 것이 0번 이상 반복된다

while문

<expr> → <term> {+ <term>}
<term> → <factor> {* <factor>}
<factor> → number | (<expr>)

- [핵심 개념]
  [ ] : 0번 혹은 1번 (optional)
  { } : 0번 이상 반복

[언어 S 문법 2:EBNF]

구체적인 구현

<stmt> → id = <expr>;
| ' {' {<stmt>} ' }' : S → S;S
| if (<expr>) then <stmt> [else <stmt>]
| while (<expr>) <stmt>
| read id;
| print <expr>;

여기서 잘못되면 가독성 아예 X!!

수식 <expr> → <bexp> {&' <bexp> | '|' <bexp>} | !<expr> | true | false

비교 <bexp> → <aexp> [<relop> <aexp>]
<relop> → == | != | < | > | <= | >= → 비교연산자

산술 <aexp> → <term> {+ <term> | - <term>}
<term> → <factor> {* <factor> | / <factor>}
<factor> → [-] (number | '('<aexp>')' | id)

우선순위

구문 다이어그램

- 구문 다이어그램
  - 각 생성규칙을 다이어그램으로 표현
  - 넌터미널 => 사각형
  - 터미널 => 원
  - 순서 => 화살표

- 수식 문법 EBNF

<expr> → <term> {+ <term> | - <term>}

<term> → <factor> {* <factor> | / <factor>}

<factor> → number | (<expr>)

구문 다이어그램

- 수식 문법 EBNF

<expr> → <term> {+ <term> | - <term>}
<term> → <factor> {* <factor> | / <factor>}
<factor> → number | (<expr>)

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

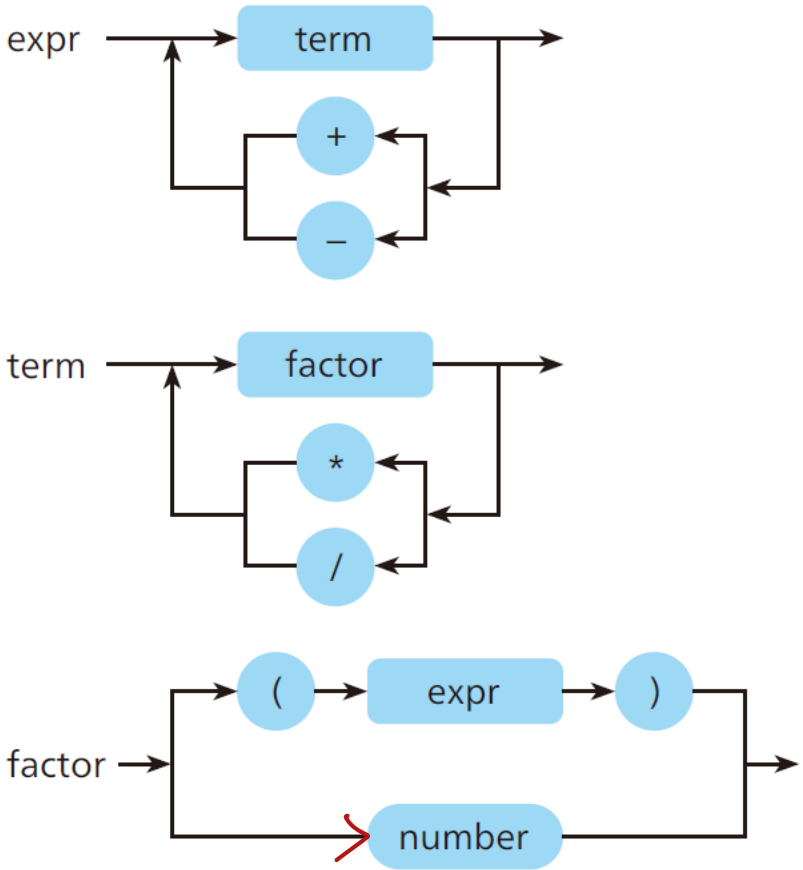


그림 2.6 수식 문법의 구문 다이어그램

2.5 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 넌터미널 → 상호순환적인 프로시저로 변환
  - 하나의 프로시저(함수,메쏘드)를 구현한다.

- 프로시저 내에서
  - 생성규칙 우변을 적용하여 좌우선 유도 하도록 작성한다.
    - <A> → <B> c <D>

      문법이기준

      ⟶ 그대로옮기기

```
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

- <expr> → <term> {+ <term>}

```
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

- <expr> → <term> {+ <term>}

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

뒤에 while문 하나 더 붙이기 X
무조건 + → - 하는게 아니니까.

수식 값 계산기

- 항의 값 계산
  - 재귀-하강 파싱 하면서 동시에 항(term)의 값을 계산

- <term> → <factor> {* <factor>}

```
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
  - <factor> → <number> | (<expr>)
  - <number> → <digit> {<digit>}
- 사용 예

```
>> 12+33
45
>> 3*5+10
25
>> (2+3)*12
60
>> 2+3*12
38
```

실습 #1: 파서/계산기

우리가 구현하는 언어에서는 integer type에 ! 연산 불가

!3+6==10

false

true

- 재귀-하강 파서/계산기 확장 구현
  - 뺄셈(-), 나눗셈(/) 추가
  - 비교연산(==, !=, >, <, !) 추가
  - 논리 연산(&, |, !)을 추가
  - Java로 작성

실습 5개
1: 1주일
2~5: 2주일씩 → 8주
누적되는 형식.

- 문법(EBNF)

확인 먼저 ~~> 이후

3 <expr> → <bexp> {& <bexp> | '|' <bexp>} | !<expr> | true | false

2
<bexp> → <aexp> [<relop> <aexp>]
<relop> → == | != | < | > | <= | >=

1
<aexp> → <term> {+ <term> | - <term>}
<term> → <factor> {* <factor> | / <factor>}
<factor> → [-] (<number> | (<aexp>))
<number> → <digit> {<digit>}