프로그래밍 언어론
-구문법

서형준
seohyoungjoon@gmail.com

2. PL(프로그래밍 언어)구문법

강의 목표:

구문법(Syntax)
- 구성요소를 이용하여 문장/프로그램을 구성하는 방법

의미 (Semantics)
- 문장/프로그램의 의미, 이것은 이런 의미
- 프로그래밍 언어 공부하는데 필요한 기본기
  . 프로그램 언어(c 파이선 등)별로 의미가 다르다.

2. PL(프로그래밍 언어)구문법

Syntax (구문)
Syntax는 문장의 형식적 구조, 즉 **문법적 규칙**을 의미. 문법에 맞는 문장을 만들어야 구문이 올바른 문장

Syntax 예제:
올바른 구문: "나는 밥을 먹는다.“ (주어(나) + 목적어(밥) + 서술어(먹는다) 정상 구문)
잘못된 구문: "나는 먹는다 밥을.

Semantics (의미론)
Semantics는 문장의 **의미**로 구문이 올바르더라도 문장의 의미가 모호하거나 논리적으로 맞지 않을 수 있다.
Semantics 예제:
의미적으로 올바른 문장: "나는 밥을 먹는다."
의미적으로 잘못된 문장: "밥이 나를 먹는다."

요약
Syntax(구문)는 문장이 문법적으로 올바르게 구성되었는지 여부를 나타내며, 한국어의 경우 주어, 목적어, 서술어의 순서가 중요
Semantics(의미론)는 문장의 의미가 논리적이고 일관성 있는지를 평가. 구문이 맞더라도 의미가 비논리적이면 의미론적 오류가 발생

따라서 **Syntax**는 형식을, **Semantics**는 내용을 다루는 개념

2. PL(프로그래밍 언어)구문법

- BNF(BNF는 **Backus-Naur Form**의 약자)
  - 보통 **"배커스-나우어 형식"** 또는 "배커스-나우어 폼“

- 이 이름은 이 형식을 정의한 두 명의 컴퓨터 과학자, **존 배커스(John Backus)**와 **피터 나우어(Peter Naur)** 의 이름에서 유래)

<BNF 구성 요소>

1. expr
   - expr은 일반적으로 expression(표현식) 줄임말
   - expr는 수학적 또는 논리적 연산을 나타내는 식(expression)을 의미. 프로그래밍 언어나 문법에서 expr은 종종 숫자, 변수, 연산자 등을 결합한 표현식을 정의하는 데 사용
     - 숫자
     - 변수
     - 연산자
     - 표현식 : 수학적, 논리적의 의미

2. PL(프로그래밍 언어)구문법

2. ::= (정의 연산자) → C 와 Python 에서 사용하는 = 와 다름

BNF (Backus-Naur Form)와 같은 문법에서 사용하는 **정의 연산자**
이 기호는 "이것은 다음과 같이 정의된다"라는 의미

역할:

::= 는 왼쪽의 비터미널(non-terminal) 기호가 오른쪽의 규칙들로 **정의**
즉, 왼쪽의 비터미널이 오른쪽의 구성요소들로 **대체**될 수 있다는 의미

예:

BNF 문법에서 자주 볼 수 있는 구문

<expr> ::= <term> ① | <term> "+" <expr> ② | <term> "-" <expr> ③

여기서 <expr>는 비터미널 기호이며, 이 기호는 다음과 같은 규칙으로 **정의**
::=는 <expr>이 오른쪽에 있는 규칙들로 대체될 수 있음

- <expr>는 단순히 <term>
- <expr>는 <term> + <expr>
- <expr>는 <term> - <expr>

2. CFG 와 BNF

CFG (Context-Free Grammar)와 BNF (Backus-Naur Form)는 프로그래밍 언어의 구문을 정의하는 데 사용되는 표현법

Context-Free Grammar (CFG)는 문법의 구조를 정의하는 형식론으로서, 프로그래밍 언어, 자연 언어, 데이터 구조 등의 문법을 분석하고 생성하기 위해 사용

- 비종단 기호(Non-terminal symbols): 더 분해될 수 있는 규칙의 왼쪽에 위치
- 종단 기호(Terminal symbols): 실제 언어의 문자나 토큰
- 생산 규칙(Production rules): 비종단 기호를 다른 비종단 기호 또는 종단 기호의 시퀀스로 변환하는 규칙
- 시작 기호(Start symbol): 문법의 시작점을 나타내는 특별한 비종단 기호

CFG는 특정 문맥에 구애 받지 않고 생성 규칙을 적용할 수 있기 때문에 '문맥 자유'라는 이름이 붙어.

.

2. CFG 와 BNF 재귀적 표현.

Backus-Naur Form (BNF)

- BNF는 CFG를 표현하기 위한 표기법 중 하나
- CFG의 규칙을 쓰기 위한 구체적인 문법
- BNF는 컴파일러 설계 및 컴퓨터 언어 설계에서 널리 사용

다음과 같은 형식을 따릅니다
:<symbol> ::= expression: 여기서 ::=는 '정의된다(defines)'를 의미

표현식은 종단 기호와 비종단 기호의 조합으로 구성될 수 있으며,
. 대체(Alternatives)를 나타내기 위해 | 기호

CFG는 문법의 구조를 정의하는 규칙의 집합입니다. BNF는 이러한 규칙을 표현하기 위한 구문적 형식을 제공합니다
, 모든 BNF는 CFG의 한 형태로 볼 수 있지만, 모든 CFG가 BNF로 표현될 필요는 없습니다.BNF는 CFG를 구현하고 문서화하는 데 유용한 도구

2. PL(프로그래밍 언어)구문법

'재귀적 표현' 이 무한한 구문을 유한한 구문으로 표현

이진수 구문 (N ::= D | ND)

D -> 0, 1 이면 이진수 N 은 D 또는 ND 이다.
=> 0, 1, 01, 101, 1111, 등 많은 것이 가능해진다.

<파이선에서 이진수를 표현할 때 쓰는 정의되어 있는 BNF 구문>
<binary-number> ::= " 0b " <binary-digits> | " 0B " <binary-digits>
<binary-digits> ::= <binary-digit> | <binary-digits> <binary-digit>
<binary-digit> ::= " 0 " | " 1"

< C 언어는 이진수를 표현하는 BNF 구문이 없음, 8진수, 16진수>
<hexadecimal-constant> ::= "0x" <hexadecimal-digit>+ | "0X"
<hexadecimal-digit>+ <octal-constant> ::= "0" <octal-digit>+
<hexadecimal-digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "a" | "b" | "c" | "d" | "e" | "f" | "A" | "B" | "C" | "D" | "E" | "F" <octal-digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"

2장 구문법(Syntax)

2.1 구문 및 구문법
2.2 유도
2.3 모호성
2.4 BNF와 구문 다이어그램
2.5 재귀 하강 파싱

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, 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

$$N \rightarrow D \qquad N \rightarrow D \mid ND$$
$$N \rightarrow ND$$

이진수: 구문법과 의미론

이진수 구문법	이진수의 의미 : 십진수 값
D → 0	V('0') = 0
| 1	V('1') = 1
or	
N → D	V(D)
| ND	V(ND) = V(N) * 2 + V(D)

↳ 자릿수 변화로 인한 값의 두배 증가를 반영하기 위함

- 101

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

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

십진수: 구문법과 의미론

$D \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$

D → 0~9

$N \rightarrow D \mid ND$

$V(\text{'0'}) = 0$
$V(\text{'1'}) = 1$
$V(\text{'2'}) = 2$
$\ldots$
$V(\text{'9'}) = 9$

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

↳ 자릿수 변화로 인한 값의 10배증가를 반영하기 위해

- 386

$V(\text{'386'}) = 386$

38 × 10 + 6
= 380 + 6
= 386

십진수: 구문법과 의미론

구문론

D → 0
| 1
| 2
...
| 9

N → D
| ND

V('0') = 0 ➔ **의미론**

- 제시된 내용 " V('0') = 0 " 는 의미론(semantic)에서의 변수나 표현식의 값을 정의하는 방법

여기서 V는 값 함수(Value function)를 나타내며, 이 함수는 언어의 구성 요소(예를 들어, 리터럴, 변수, 표현식 등)에 구체적인 값을 할당

- **해석**
- **V('0')**: 이는 문자 또는 리터럴 '0'의 값을 나타내는 함수 호출입니다.
- **= 0**: 이는 '0'이라는 문자 또는 리터럴이 실제로 가지는 값이 0임을 나타냅니다

수식의 구문법

- 일반적인 수식의 구문론

- 수식 의 예제 (+ 와 * 하기가 가능한 수식)

  5, 5 + 13, 5 + 13 + 4, 5 * 13 + 4, (5 + 13), (5 + 13) * 12, ...

- 구문법 : 쓰는 방법(-> 형태로 작성될 수 있다"는 의미)

  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	수식의 의미(시맨틱스)
$E \rightarrow E * E$	V(E * E) = V(E) * V(E)
$\mid E + E$	V(E + E) = V(E) + V(E)
$\mid (E)$	V((E)) = V(E)
$\mid N$	V(N)

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

- 수식의 의미

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

- 문장 S의 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 → id = E
  | if E then S else S
  | while (E) S

문맥-자유 문법 CFG

- 문맥-자유 문법 CFG는 다음과 같이 구성
  - 터미널 심볼의 집합 T (숫자, 문자, 연산자 등)
  - 넌터미널 심볼의 집합 N
  - 시작 심볼 S (넌터미널 심볼 중에 하나)
  - 다음과 같은 형태의 생성(문법) 규칙들의 집합

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

    $X \rightarrow \varepsilon$ (오른쪽이 빈 스트링인 경우 ε (epsilon, 빈 문자열) 대체)
- 보통 넌터미널(nonterminal) 심볼은 대문자로,

  터미널(terminal) 심볼은 소문자로 표기한다.

생성 규칙

- 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 의미

- 문장 S

  S → id = E 선언문

  S → if E then S else S 조건문

  S → while (E) S 반복문

S → id = E

  | if E then S else S

  | while (E) S

[언어 S의 문장 요약 문법 1]

```
Stmt S → id = E
       | S; S
       | if E then S
       | if E then S else S
       | while (E) S
       | read id
       | print E
```

완성인 Stmt는 이렇게 표현이 가능하고

```
Expr E → n | id | true | false
       | E + E | E - E | E * E | E / E | ( E )
       | E == E | E != E | E < E | E > E | !E
```

여기 다 완료되어야 이제 되는거지?

표현식인 Expr은 이런식 (Page 16)처럼 된다는 말인가? 그래서 다 마니라 괄호 () 쓸수 있다는거?

[언어 S의 문장 요약 문법 1]

CFG로 작성

Stmt S → id = E

| S; S | if E then S

| if E then S else S

| while (E) S

| read id | print E

id는 변수 식별자이고, E는 표현식(expression)

S → S; S → 두 개의 명령문 S를 세미콜론(;)으로 연결하여 순차적 실행

S → if E then S: E가 참일 때 명령문 S 실행

S → while (E) S: E가 참인 동안 명령문 S를 반복

S → read id: 입력을 받아 변수 id에 저장

S → print E: 표현식 E의 결과를 출력

[언어 S의 문장 요약 문법 1]

언어 이해

CFG로 작성

Expr E → n | id | true | false

| E + E | E – E | E * E | E / E | (E)

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

n: 숫자 리터럴 예를 들어, 42, 3.14 등

id: 식별자(변수 이름) x, temperature 등

true, false: 불리언 리터럴, 논리적 참과 거짓표현

E + E: 두 표현식의 합 E – E: (차) E * E: (곱) E / E: (나누기)

(E): 괄호 안의 표현식을 그룹화 우선순위 명시

E == E: 두 표현식이 같은지 비교

E != E: 두 표현식이 다른지 비교

E < E: E > E: !E: 표현식 E의 논리적 부정

[언어 S의 문장 요약 문법 1]

BNF로 작성

(STATEMENT)

<S> ::= <id> "=" <E> | <S> ";" <S> | "if" <E> "then" <S> | "if" <E> "then" <S> "else" <S> | "while" "(" <E> ")" <S> | "read" <id> | "print" <E>

(Expression)

<E> ::= <n> | <id> | "true" | "false" | <E> "+" <E> | <E> "-" <E> | <E> "*" <E> | <E> "/" <E> | "(" <E> ")" | <E> "==" <E> | <E> "!=" <E> | <E> "<" <E> | <E> ">" <E> | "!" <E>

2.2 유도

유도(Derivation)

- 입력된 문장 혹은 프로그램이 문법에 맞는지 검사하는 것을 구문검사라고 한다.

어떤 문장 혹은 프로그램이 구문법에 맞는지는 어떻게 검사할 수 있을까?

- 입력된 문장이 문법에 맞는지 검사하려면 문법으로부터 유도(derivation)해 보아야 한다.

- **[핵심 개념]** 어떤 문장이 문법으로부터 유도 가능하면 문법에 맞는 문장이고 그렇지 않으면 문법에 맞지 않는 문장이다. ⇒ 해당 내용을 할 수 있어야함.

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.

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

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

유도(Derivation)

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

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

⇒ X_i *가* $Y_1\ Y_2 \ldots Y_n$ *로 한번 대체*

⇒ *생성 규칙이 A → BC인 경우, 문자열 xAy에서 A를 BC로 한 번에 대체하여 xBCy*

- 유도(Derivation) ⇒* 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 – 숙지 필요(^^) : 9월 11일 ~

- 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

- 3 + 4 + 5 유도 ?

고유페이지가 없음.

좌측 유도와 우측 유도

- 좌측 유도(leftmost derivation)
  - 각 직접 유도 단계에서 가장 왼쪽 넌터미널을 선택하여 이를 대상으로 생성 규칙을 적용한다.
  - 3 + 4 * 5의 좌측 유도

    E => E + E ⇒ E + E * E ⇒ N + E * E ⇒ 3 + E * E => 3 + N * E

    => 3 + 4 * E => 3 + 4 * N => 3 + 4 * 5

- 우측 유도(rightmost derivation)
  - 각 직접 유도 단계에서 가장 오른쪽 넌터미널을 선택하여 이를 대상으로 생성 규칙을 적용하면 된다.
- 3 + 4 * 5의 우측 유도
  - E ⇒ E + E ⇒ E + E * E ⇒ E + E * N ⇒ * E + E * 5

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

문법 G 언어 : 구문법을 정하고 사용가능 언어

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

- 예: 문법 G(새로 만든 언어 G 의 구문은 괄호와 a로 선언)
  - S → (S)
  - S → a

(1) 먼저 몇 개의 가능한 스트링을 생성해 보면 다음과 같이 무한대.

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

(2) 이들을 집합 형태로 표현해 보자. : 괄호가 n번 인 집합, 무한대

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

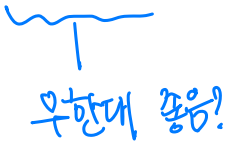
문법 G 언어 : 구문법을 정하고 사용가능 언어

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

예제: 2진수 덧셈만 가능한 문법과 언어 L(G)는 ?

D -> 0 | 1

N-> D | ND, L(G) = { 0, 1, 00, 11, 111, …} 무한대 가 가능



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.

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

- 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.

➔ 생성된 스트링이 구문에 적합한지 확인하는 과정을 트리로 작성

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

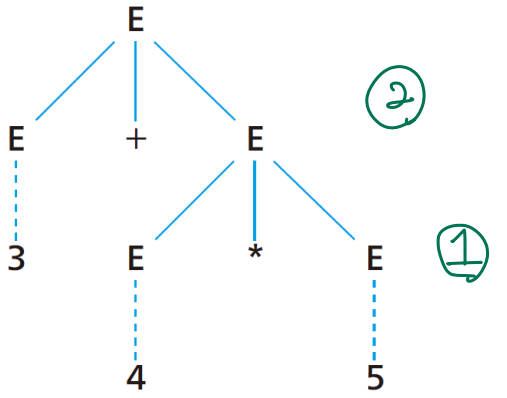


그림 2.1 3 + 4 * 5를 위한 파스 트리

유도 트리에 대한 참조

- 이 트리 구조는 3 + (4 * 5)와 같은 결합 성질을 보여준다.

위 페이지 유도식 결과 = 28

- 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.

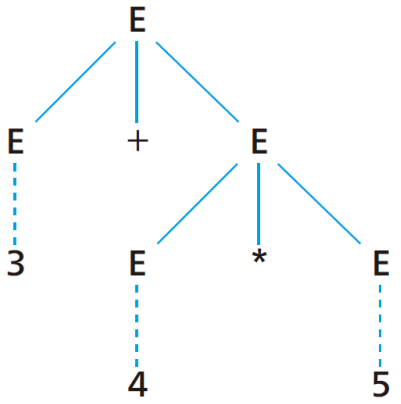


그림 2.2 3 + 4 * 5의 첫 번째 파스 트리

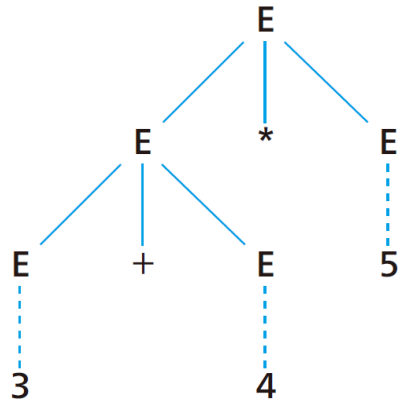


그림 2.3 3 + 4 * 5의 두 번째 파스 트리

- 모호성은 나쁘다
  - 왜 ?

모호성 처리 방법 1

- 문법 재작성
  - 원래 언어와 같은 언어를 정의하면서 모호하지 않도록 문법 재작성

- 구조식이 주어지면 구조식에 대한 유도식 & 트리 알아야함.

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

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow N \mid (E)$

→ 곱 다는 뭐임?: 터미널 단위의 정합

→ N이 생략된 것. 내가 한 땐 둘이서!!

그러나 → 해당 구조식은 주어짐. 주어진 것은 트리로 다 만들 수 있음

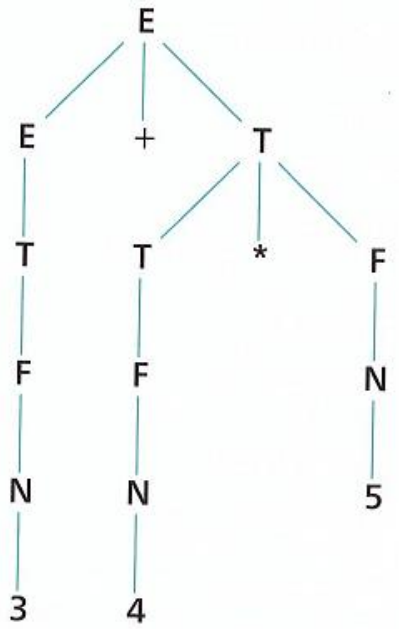


그림 2.4 3 + 4 * 5에 대한 유도 트리

23

- 3 + 4 * 5의 좌측 유도
  - $E \Rightarrow E + T \Rightarrow^* N + T \Rightarrow 3 + T \Rightarrow 3 + T * F \Rightarrow 3 + F * F$
    $\Rightarrow 3 + N * F \Rightarrow^* 3 + 4 * N \Rightarrow^* 3 + 4 * 5$ => 23

모호성 예: The Dangling Else

- 모호한 문법

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

S가 어느 S에 걸려야 하는지 확인X

- 이 문장에 대한 두 개의 파스 트리

if e1 then if e2 then s1 else s2

1-1 1-2

해당 S가 1-1의 else인지 1-2의 else인지 모름.

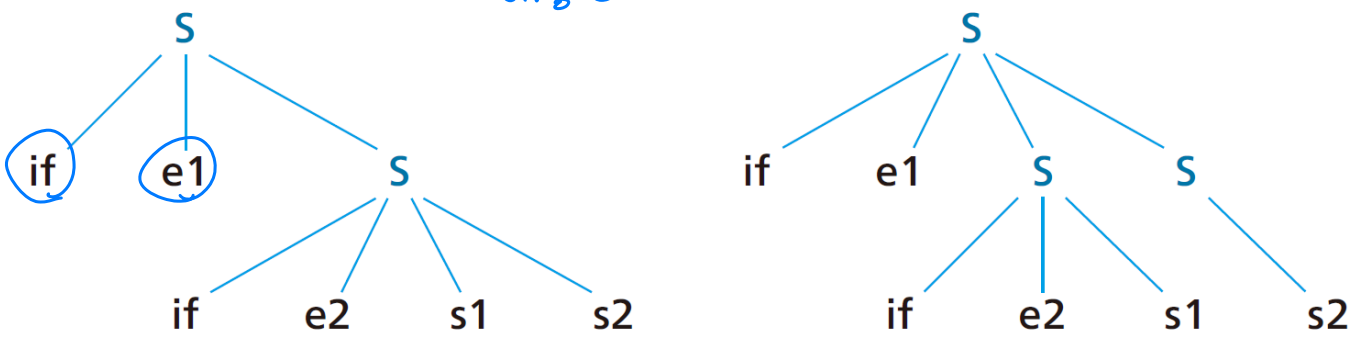


그림 2.5 if 문: 모호한 문법

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

2.4 BNF와 구문 다이어그램

BNF/EBNF

- BNF(Backus-Naur Form)

  <expr> → <expr> + <term> | <term>

  <term> → <term> * <factor> | <factor>

  <factor> → number | (<expr>)

- EBNF(Extended BNF)

  <expr> → <term> {+ <term>}

  <term> → <factor> {* <factor>}

  <factor> → number | (<expr>)

- [핵심 개념]

  [ ] : 0번 혹은 1번 (optional) — 선택

  { } : 0번 이상 반복

[언어 S 문법 2:EBNF]

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

<expr> → <bexp> {& <bexp> | ‘|’ <bexp>} | !<expr> | true | false
<bexp> → <aexp> [<relop> <aexp>]
<relop> → == | != | < | > | <= | >=
<aexp> → <term> {+ <term> | - <term>}
<term> → <factor> {* <factor> | / <factor>}
<factor> → [-] (number | ‘(’<aexp>‘)’ | id)

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.

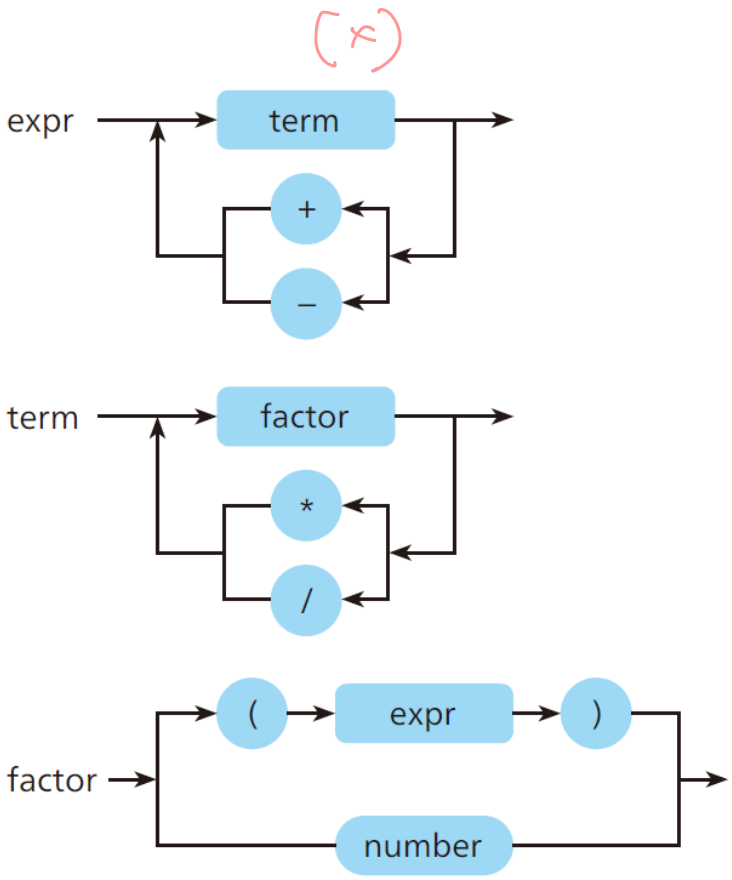


그림 2.6 수식 문법의 구문 다이어그램

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

<A> → <B> c <D>

```
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;
}
```

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
```

파서/계산기

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

- 문법(EBNF)

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

  <bexp> → <aexp> [<relop> <aexp>]

  <relop> → == | != | < | > | <= | >=

  <aexp> → <term> {+ <term> | - <term>}

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

  <factor> → [-] (<number> | (<aexp>))

  <number> → <digit> {<digit>}