3장 구문과 의미론

서론

- 구문(syntax)
  - 식, 문장, 프로그램 단위의 형태 또는 구조
  - 문법(grammar)로 표현
- 의미론(semantics)
  - 식, 문장, 프로그램 단위의 의미
- 구문과 의미론은 언어의 정의를 제공
  - 언어 정의의 사용자
    - 언어 설계자들
    - 언어 구현자들
    - 언어 사용자들
  - 범용적 구문 표기법(즉, 문법)이 존재하나, 의미론의 경우 아직 부재

용어: lexeme, token, id

- 어휘항목(lexeme)
  - 토큰을 이루는 문자열
- 토큰(token)
  - 의미적으로 구분되는 최소 단위
  - Ex. index = 2 * count + 17;
- id(identifier)
  - 문자열(lexeme)이 변수명, 함수명, 클래스명 등을 이룰 때
  - Ex. index, count

id의 문법적 표현

- id(identifier)를 만드는 규칙

<id> -> <letter> (<letter> | <digit>)*
<letter> -> a | b | c | …… | z
<digit> -> 0 | 1 | 2 | …… | 9

- 파생(derivation)

<id> => <letter> (<letter> | <digit>)*
=> i (<letter> | <digit>)*
=> i n (<letter> | <digit>)*
=> i n d (<letter> | <digit>)*
=> i n d e (<letter> | <digit>)*
=> i n d e x

문법의 활용

- 언어 인식기
  - 언어 L의 알파벳으로 구성된 입력 문자열을 읽어들여서 L에 속하는지를 판단(accept/reject)하는 인식 장치
- 언어 생성기
  - 언어의 문장들을 생성하는 장치
  - 주어진 문법에 의해 언어가 파생(derivation)된다.
- 언어 생성기와 인식기간의 관계
  - 형식언어와 컴파일러 설계 이론의 연구 성과
- 프로그래머가 생성한 문장이 올바른지를 판단하는 방법은?

문법의 활용

- 언어 인식기
  - 언어 L의 알파벳으로 구성된 입력 문자열을 읽어들여서 L에 속하는지를 판단(accept/reject)하는 인식 장치

  - Ex. 언어(L) = 0(10)*
    입력 알파벳 = {0, 1}
    입력 문자열 = 01010

    질문) 01010 은 0(10)* 에 속하는가?

  - 언어(L)에 대한 automata를 그려서 accept 여부를 판단

문법의 활용

- 언어 생성기
  - 언어의 문장들을 생성하는 장치
  - 주어진 문법에 의해 언어가 파생(derivation)된다.

  - Ex. 다음 문법은 어떤 언어를 생성하는가?

    S -> S10 | 0

  - 답) S => 0
    또는 S => S10 => 010
    또는 S => S10 => S1010 => S1010 ……10 => 01010 ……10
    => 0(10)*

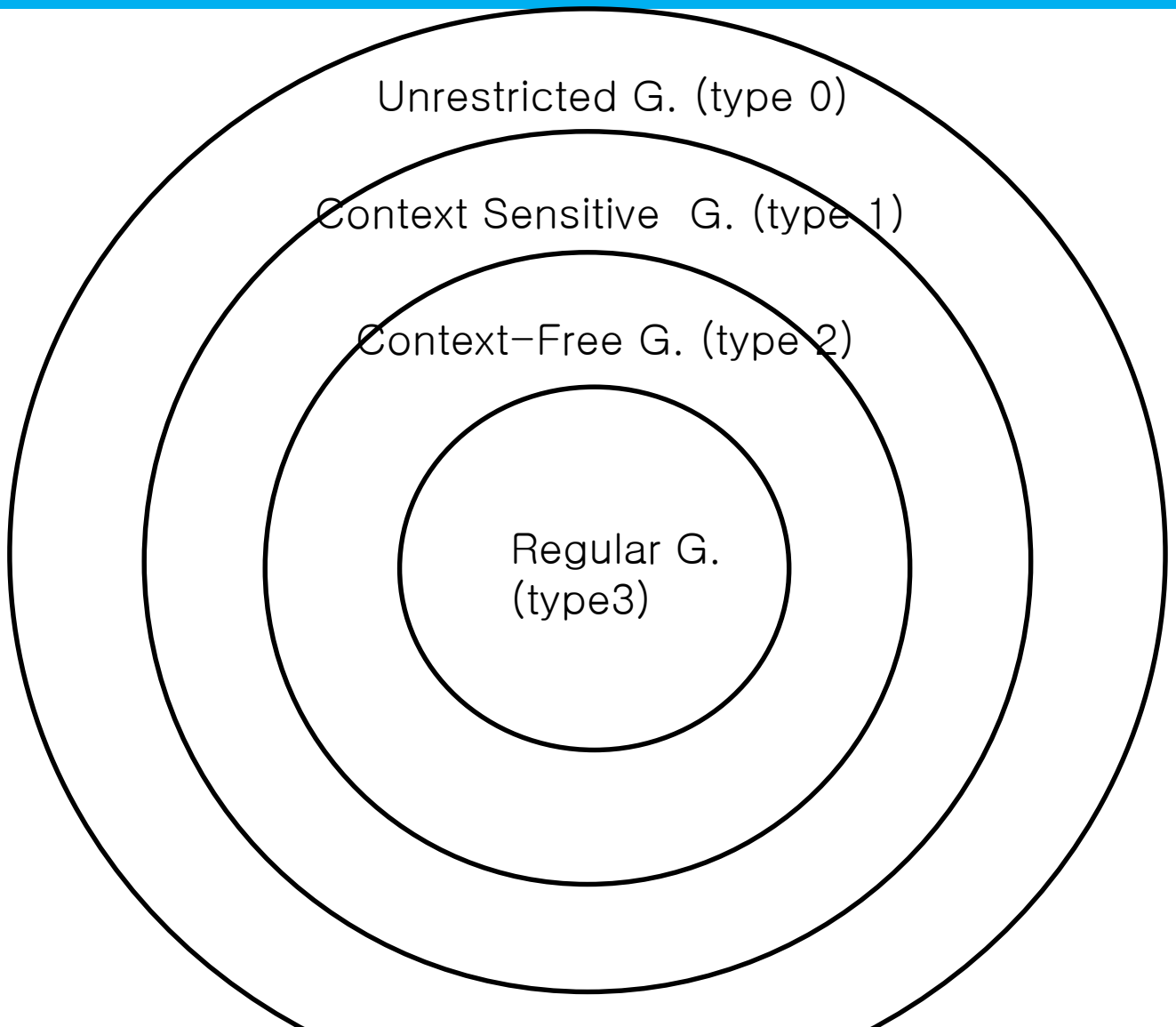
    따라서 L = {0, 010, 01010, ……, 01010……10}
    = 0(10)*

문법의 중요성이 대두

- 문맥자유 문법(CFG: context-free grammars)
  - 1950년대 중반, Noam Chomsky에 의해서 개발
  - 자연 언어에 대한 구문 기술 목적으로 4가지 유형 언어 정의 (Chomsky의 Hierarchy)
  - 프로그래밍 언어 기술에 유용한 2가지 유형
    - 정규 문법(regular grammars) (type 3)
    - 문맥자유 문법 (type 2)

- BNF(BNF: Backus-Naur Form)(1959)
  - John Backus가 고안 (Algol 58 기술)
  - 이를 Peter Naur가 수정(Algol 60 기술)
  - BNF는 문맥자유 문법과 동일

정규문법과 CFG


Unrestricted G. (type 0)
Context Sensitive G. (type 1)
Context-Free G. (type 2)
Regular G.
(type3)

CFG

- Context-Free Grammar(CFG)의 특징

  A -> β 여기서, A는 nonterminal의 집합,
  β는 nonterminal과 terminal이 섞여
  나오는 arbitrary string
  (단, empty string은 허용치않음)

- nonterminal의 집합 = {A, B, C, …… Z}
  terminals는 그 외의 소문자 및 특수문자

- Ex. E -> E + E
  E -> E * E
  E -> (E)
  E -> id

Regular Grammar(정규문법)

- 정규문법의 특징

A -> βB | β ; right-linear G.
혹은 A -> Bβ | β ; left-linear G.

여기서, A와 B는 최대 하나의 nonterminal,
β는 0개 이상의 terminal로 구성 (즉, empty도 허용)

- Ex 1) right-linear
  S -> 0A
  A -> 10A | ε

Ex 2) left-linear
S -> S10 | 0

- Ex 1)과 Ex 2)는 같은 string 0(10)* 를 생성한다.

BNF (CFG과 기능상 동일, 표기법만 다름)

- 규칙 표현
  - LHS, RHS로 구성
    - LHS(left-hand side): 한 개의 논터미널 (nonterminal symbol)로 표현 (nonterminal을 BNF에서는 각괄호로 나타내고, CFG 에서는 대문자로 나타냄)
    - RHS(right-hand side): 터미널(terminals)과 논터미널들로 구성된 문자열로 표현
    - 터미널은 어휘항목 또는 토큰을 이룸 (terminal을 BNF에서는 각괄호 없이 나타내고, CFG에서는 소문자 혹은 특수문자로 나타냄)
  - Ex.

```
<id_list> → id | id,<id_list>
<if_stmt> → if <logic_expr> then <stmt>
```

BNF 기본 사항

- 한 개 이상의 생성규칙들로 구성
- 시작 기호(start symbols)은 문법에 포함된 특정 논터미널 기호
- 문법 G는 다음 4가지로 구성되나, 보통은 P로만 나타낸다

G = (N, T, S, P)

N: 논터미널들의 집합(a set of terminals)

T: 터미널들의 집합(a set of terminals)

S ∈ N: 시작 기호(start symbol)

P: 생성 규칙들의 집합(production rules)

BNF 규칙

- 두 개 이상의 RHS를 포함 가능

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

- 재귀(recursion)를 사용되기도 함

```
<id_list> → id
          | id, <id_list>
```

- 명칭리스트의 예;

  예1) x, y, z

  예2) foo1, foo2, index, result

Regular G.와 CFG

- 원래는 다음과 같이 표현해야 한다.

<id_list> → <id> | <id>, <id_list> } CFG

<id> -> <letter> (<letter> | <digit>)*
<letter> -> a | b | c | …… | z
<digit> -> 0 | 1 | 2 | …… | 9 } Regular G.

- 그러나 보통은 다음과 같이 표현한다.

<id_list> → id | id, <id_list> } CFG

- 이유?

1) <id>, <letter>, <digit> 는 regular G. 이다. 이것은 lexical analyzer에 의해 token으로 인식된다.
2) 이 token이 syntax analyzer의 input이 된다.
3) syntax analyzer에 필요한 문법은 CFG 뿐이다.
4) 즉, syntax analyzer에서는 <id>가 이미 분석되어 token(=terminal)으로 인식

Example: 문법

문법 G1

```
<program> → begin <stmt_list> end
<stmt_list> → <stmt>
              | <stmt>; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
               | <var> - <var>
               | <var>
```

- "begin A = B+C; B=C end"의 문장이 문법 G1으로부터 생성되는가?

유도(derivations)

- 문장의 생성과정을 보여주는 일련의 규칙 적용
- 방법

1. 시작 기호 S부터 시작
2. 문장형태에 논터미널 X가 포함되면, X를 LHS로 갖는 규칙을 적용
   - 유도 각 단계의 결과를 문장형태(sentential form)라 함
   - 유도 각 단계에서 단지 한 개의 규칙만을 적용
   - 전 단계의 문장형태에서 논터미널 한 개를 그 정의 중의 한 개로 대체

     Ex.

     S =>* XYZ, X → $y_1 y_2 \dots y_n$

     S =>* $y_1 y_2 \dots y_n$YZ
3. 위와 같은 규칙 적용을 반복
4. 문장형태에 논터미널이 포함되지 않으면 유도를 종료
   - 이러한 문장형태를 문장(sentence)라 한다
   - 문장은 터미널들만 포함

leftmost vs. rightmost derivation

- 최좌단 유도(leftmost derivation)
  - 각 문장형태에서 가장 좌측에 위치한 논터미널이 규칙 적용을 위해서 선택된다.

- 최우단 유도(rightmost derivation)
  - 각 문장형태에서 가장 우측에 위치한 논터미널이 규칙 적용을 위해서 선택된다.

Example: leftmost vs. rightmost derivation

문법 G1

```
<program> → begin <stmt_list> end
<stmt_list> → <stmt>
              | <stmt>; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
               | <var> - <var>
               | <var>
```

- "begin A = B+C; B=C end" 라는 문장에서
  - 위 문장 생성에 대한 최좌단 유도(leftmost derivation)를 보여라.
  - 위 문장 생성에 대한 최우단 유도(rightmost derivation)를 보여라.

Example: rightmost derivation

문법 G2

```
<assign> → <id> = <expr>
<id> → A | B |C
<expr> → <id> + <expr>
        | <id> * <expr>
        | (<expr>)
        | <id>
```

- “A = B * (A + C)”의 문장에 대한 최우단 유도를 보여라.

파스 트리(parse tree)

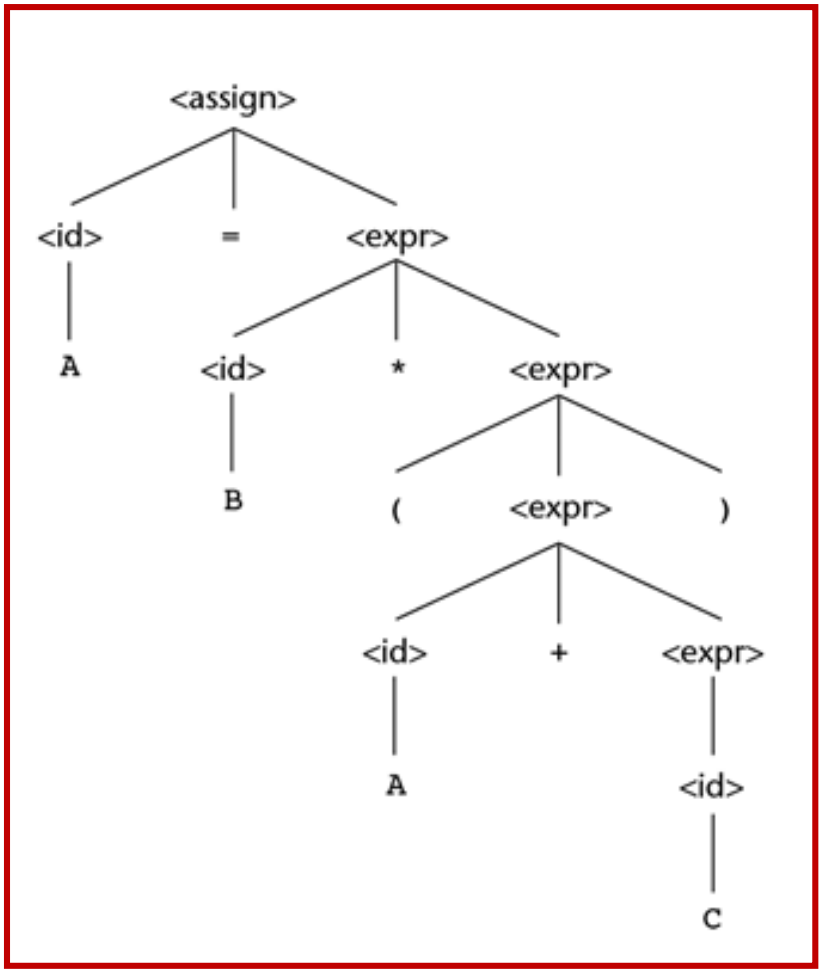
- 유도의 계층적 표현
  - 문장 “A=B*(A+C)”에 대해 유도해 보자

```
<assign> → <id> = <expr>
<id> → A | B |C
<expr> → <id> + <expr>
          | <id> * <expr>
          | (<expr>)
          | <id>
```

  - 그것을 그림으로 표현한 것이 parse tree
    - Top-down parsing
    - Bottom-up parsing

파스 트리(parse tree)

- 문장 구조에 대한 계층적 표현
  - 내부 노드: 논 터미널
  - 잎 노드: 터미널



Example: parse tree

- 다음 문장에 대한 파스 트리를 그려라.

Begin A=B+C; B=C end

```
<program> → begin <stmt_list> end
<stmt_list> → <stmt>
                    | <stmt>; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
                    | <var> - <var>
                    | <var>
```

문법의 모호성(Ambiguity)

- 정의
  - 주어진 문법 G가 2개 이상의 다른 파스 트리를 갖는 문장형태를 생성하면, G는 모호하다(ambiguous)라고 말한다.

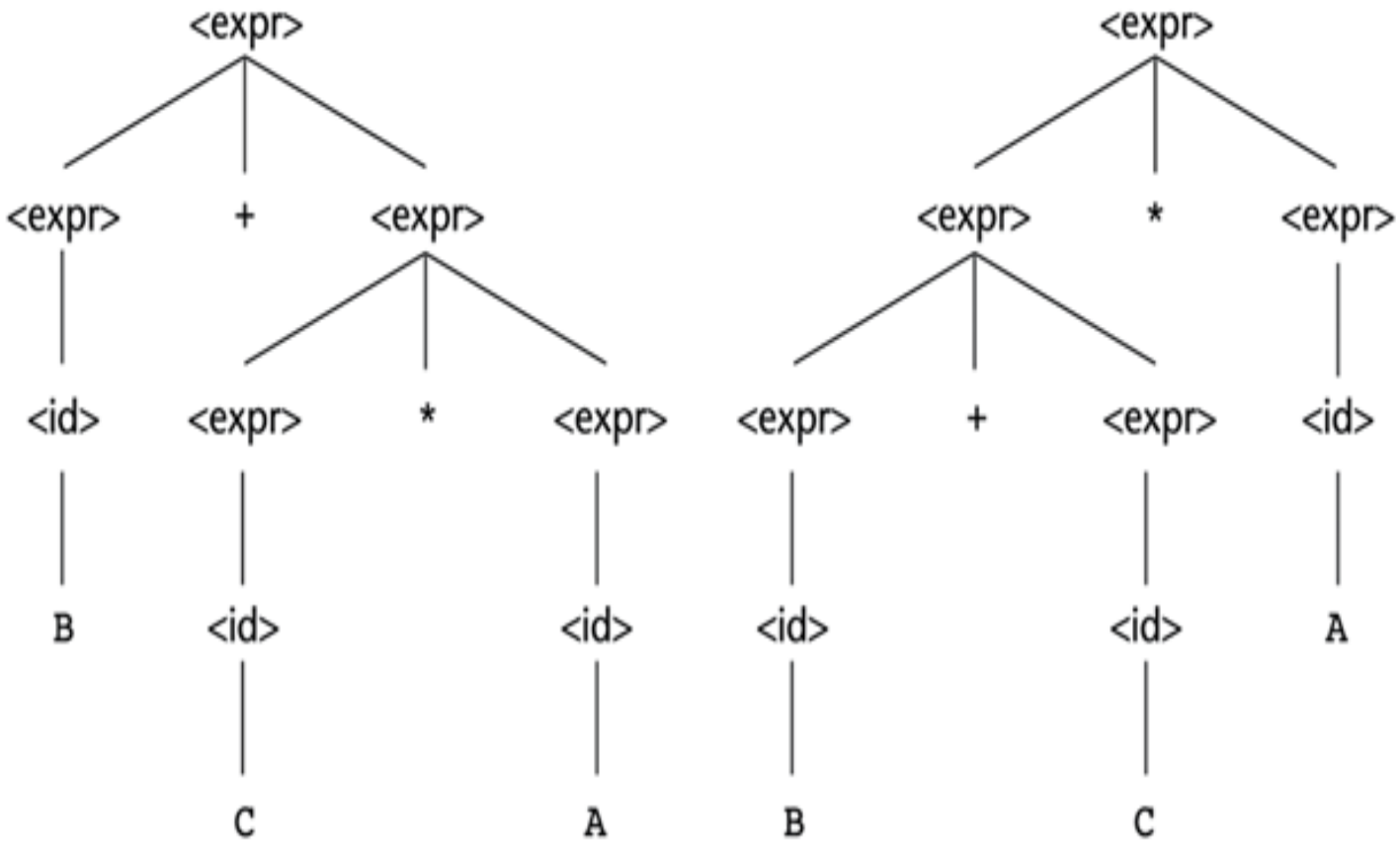
Example: Ambiguity

문법 G3

```
<expr> → <expr> + <expr>
         | <expr> * <expr>
         | (<expr>)
         | <id>
<id> → A | B |C
```

- 위의 문법 G는 모호한가?
  - "B + C * A"의 문장에 대한 parse tree를 구하라.

Example: Ambiguity



Unambiguous Grammar

문법 G3

```
<expr> → <expr> + <expr>
         | <expr> * <expr>
         | (<expr>)
         | <id>
<id> → A | B |C
```

를 종종 E → E + E | E * E | (E) | id
로 표현한다.

- 위의 문법을 Unambiguous Grammar로 고쳐라?

Unambiguous Grammar

문법 G3’

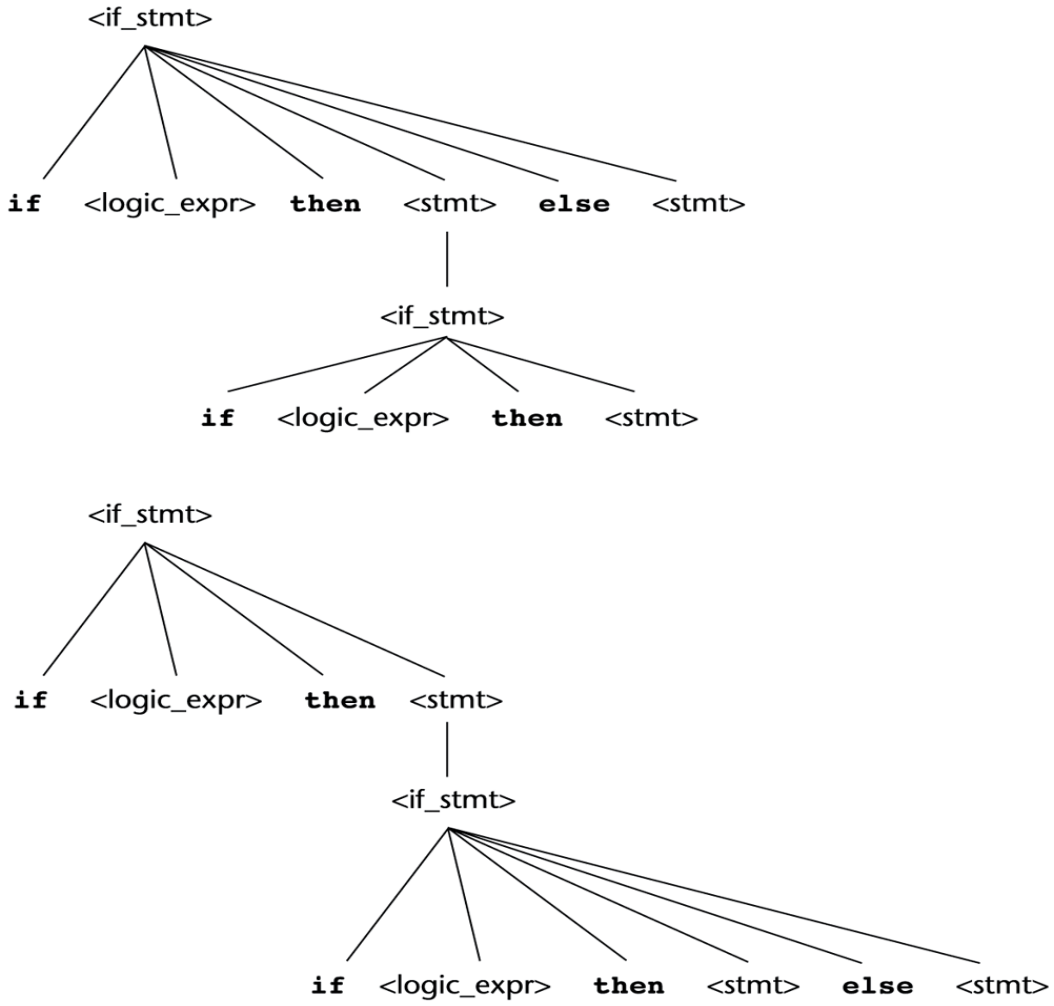
```
E → E + T | T
T → T * F | F
F → ( E ) | id
```

- “B + C * A”의 문장에 대한 parse tree를 구하라.

모호성을 갖는 다른 예제: if문

```
<if_stmt> → if <logic_expr> then <stmt>
      | if <logic_expr> then <stmt> else <stmt>
```

모호성을 갖는 다른 예제: if문



모호성을 없앤 if문

```
<stmt> → <matched> | <unmatched>

<matched > → if <logic_expr> then <matched> else <matched>
      | any non-if statement

<unmatched > → if <logic_expr> then <stmt>
      | if <logic_expr> then <matched> else <ummatched>
```

연산자 우선순위

- 문법 상에 연산자 우선순위를 부여
  - 파스 트리의 낮은 위치의 연산자가 높은 위치의 연산자 보다 높은 우선순위를 갖는다.
  - 이와 같이 파스 트리상에 연산자 우선순위가 표현되게 하는 문법을 작성하여 모호성을 제거할 수 있다.

예제: 연산자 우선순위

- 다음에서 *이 +보다 높은 우선순위가 되게 문법을 수정하라.

```
<assign> → <id> = <expr>
<id> → A | B |C
<expr> → <expr> + <expr>
       | <expr> * <expr>
       | (<expr>)
       | <id>
```

  - Hint: 새로운 논터미널 <fact>와 <term>을 도입.

예제: 연산자 우선순위

- 모호성을 없앤 문법

```
<assign> → <id> = <expr>
<id> → A | B |C
<expr> → <expr> + <term> | <term>
<term> → <term> * <fact> | <fact>
<fact> → (<expr>)| <id>
```

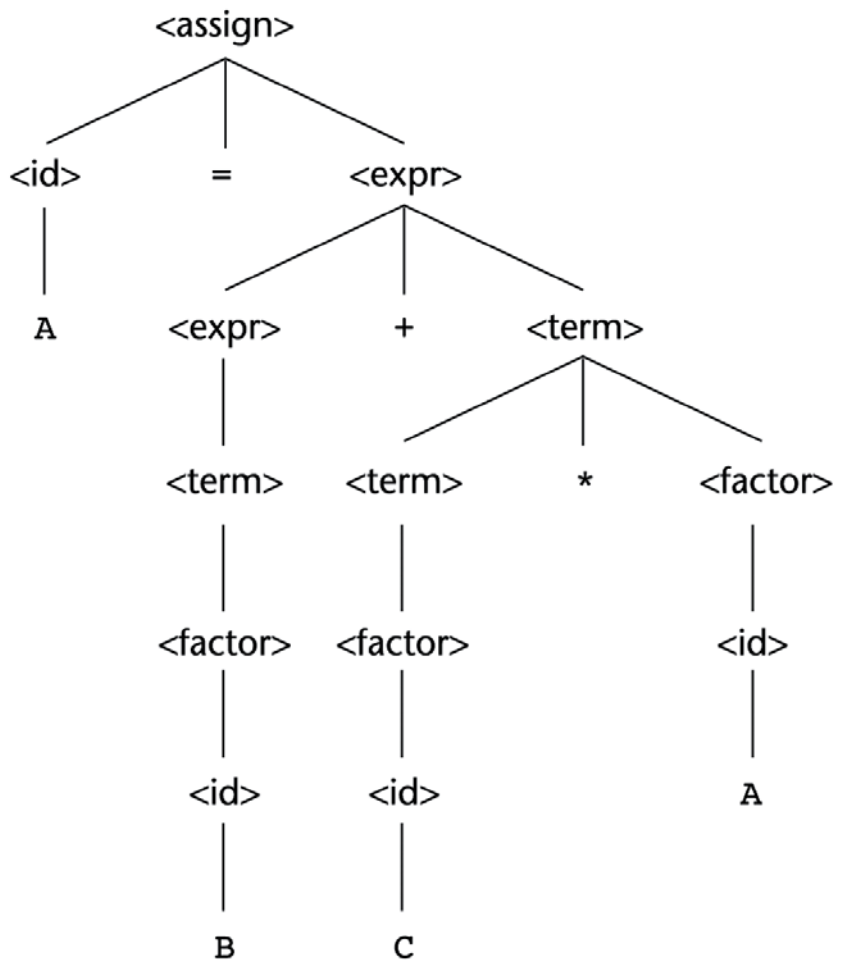
혹은

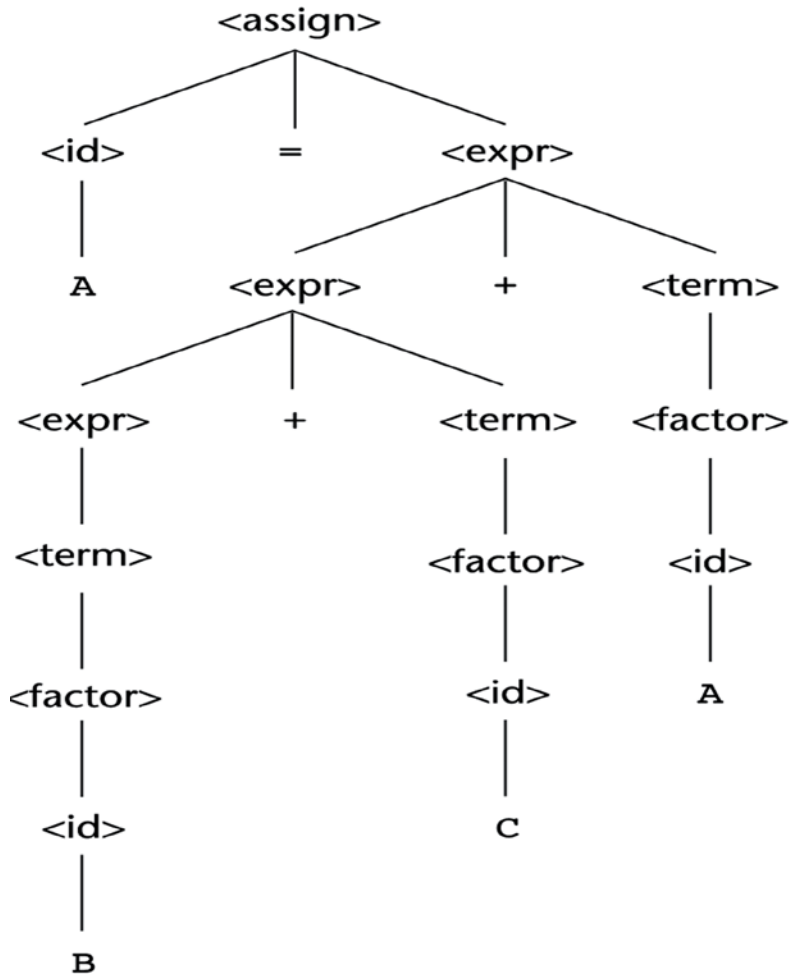
```
E → E + T | T
T → T * F | F
F → ( E ) | id
```

- "A = B + C * A", "A = B + C + A"에 대한 파스트리를 보자

Unambiguos Grammar

- 우선순위가 분명해진다. C*A를 먼저 수행 . (이유?)
- left association rule(좌결합법칙)이 적용된다. (이유?)





연산자 결합규칙(Association rule)

- 연산자 결합규칙도 문법상에 표현할 수 있다.

- Ex.
  - 다음 문법에서 각 연산자의 결합규칙은?

```
<assign> → <id> = <expr>
<id> → A | B |C
<expr> → <expr> + <term> | <term>
<term> → <term> * <fact> | <fact>
<factor> → <exp> ** <factor> | <exp>
<exp> → (<expr>)| <id>
```

  - “A ** B ** C”에 대한 파스 트리를 보자

연산자 결합규칙

- 규칙에서 LHS가 RHS의 처음에 오면, 그 규칙은 좌 순환적(left recursive)이라 한다.
- 규칙에서 LHS가 RHS의 맨 끝에 오면, 그 규칙은 우 순환적(right recursive)이라 한다.
- 좌 순환 규칙은 좌결합 법칙을 명세하고, 우 순환 규칙은 우결합 법칙을 명세한다.
- Ex.
  - 다음 문법에서 각 연산자의 결합규칙은?

```
E → E + T | T
T → T * F | F
F → P ** F | P
P → ( E )| id
```

Extended BNF(EBNF)

- BNF의 표현력을 향상시키지는 않으나 판독성/작성력을 향상시킴
- 확장사항

1. []: 선택 사항 표현
   ex. <selection> → if (<expr>) <stmt> [else <stmt>]
2. {}: 반복 사항 표현(0 or more)
   ex. <ident_list> → <ident> {, <ident>}
3. (): 다중 선택 표현
   ex. <term> → <term> (* | /) factor
4. + : 한번 이상 반복 표현
   ex. <compound> → begin {<stmt>}$^{+}$ end

예제: EBNF

- 다음 BNF를 EBNF로 표현하라.

BNF:

```
<expr> → <expr> + <term> | <expr> - <term> | <term>
<term> → <term> * <factor> | <term> / <factor> | <factor>
<factor> → <exp> ** <factor> | <exp>
<exp> → (<expr>)| <id>
```

EBNF:

```
<expr> → <term> {(+ | -)<term>}
<term> → <factor> {(* | /)<factor>}
<factor> → <exp> {**<exp>}
<exp> → (<expr>)| <id>
```

BNF, EBNF의 최근 변경사항

구분	최근 변경사항
→	:
선택사항 표현 []	아랫첨자 opt 사용 Ex. Proc → name(parameterList$_{opt}$)
다중선택표현 ()	"one of" 사용 Ex. Operator → one of + * - / < >
수직바 '|'	'|' 를 생략한 채, 각 RHS를 단순히 별도의 줄에 표현

Example: in C

statement:
for(expr-1$_{opt}$; expr-2$_{opt}$; expr-3$_{opt}$) statement
if (expression) statement
while (expression) statement
…