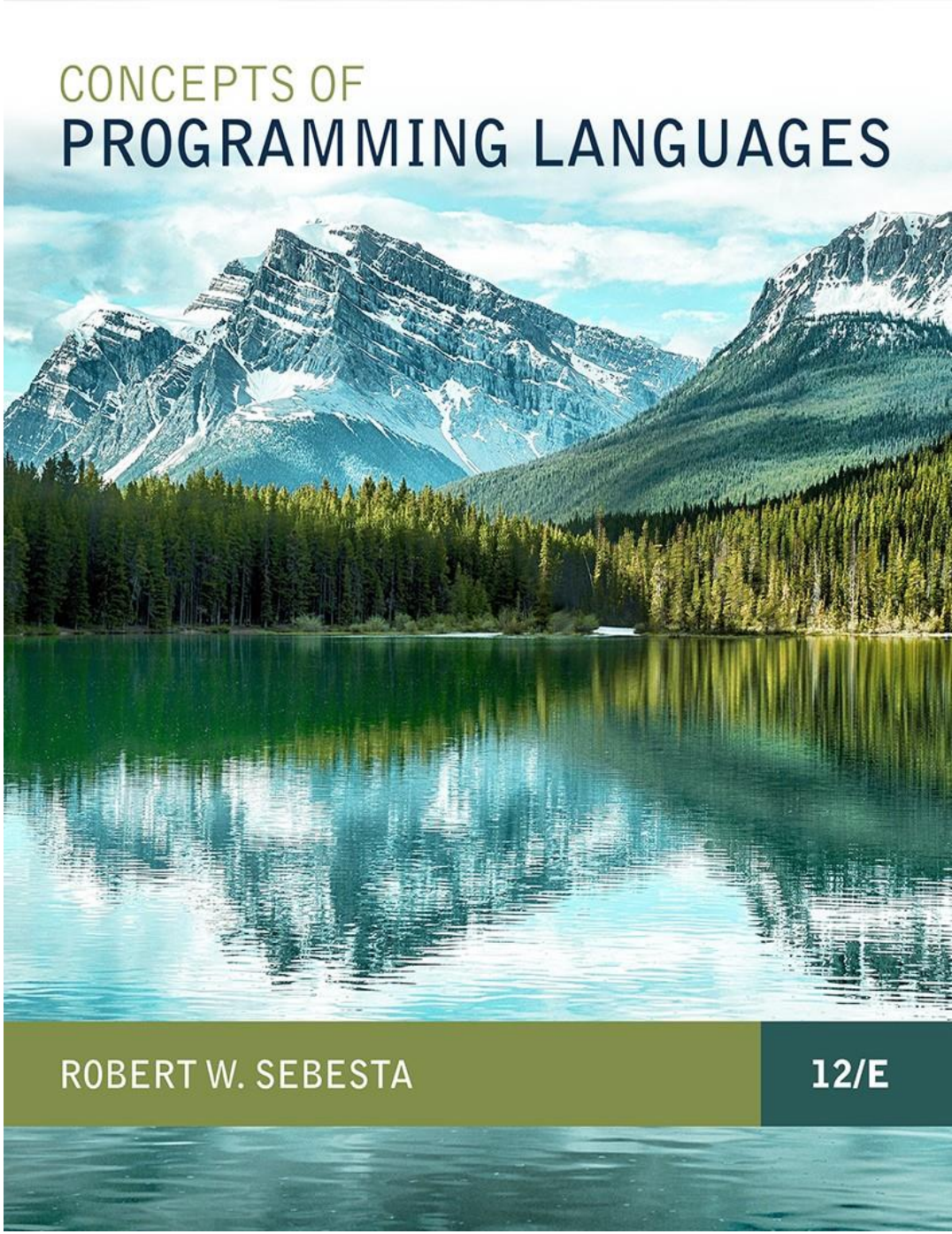
Chapter 3

구문과 의미론



ISBN 0-321-49362-1

Chapter 3 Topics

- 서론
- 구문기술의 일반적인 문제
- 구문기술의 형식적 방법
- 속성 문법
- 프로그램 의미 기술 : 동적 의미론

서 론

- **구문(Syntax)**
  : 표현식, 문장, 프로그램 단위의 형식

- **의미론(Semantics)**
  : 표현식, 문장, 프로그램 단위의 의미

- 구문과 의미론을 이용하여 언어를 정의
  –언어 정의의 사용자 : 언어 설계자, 언어 구현자, 언어사용자

(예) while문의 구문과 의미론은?
구문 : while (불리안_표현식) 문장
의미론?

구문 기술의 용어

- *문장(sentence):* 언어의 알파벳으로 구성된 문자열
- 언어(*language):* 문장들의 집합
- 어휘항목(*lexeme):* 가장 낮은 수준의 구문 단위
  (예) `*`, `sum`, `begin`
- 토큰(*token):* 어휘항목의 범주
  (예) index = 2 * count + 17;

구문 기술의 용어(continued)

(예) index = 2 * count + 17;

어휘항목	토큰
index	identifier
=	equal_sign
2	init_literal
*	mult_op
count	identifier
+	plus_op
17	init_literal
;	semicolon

언어의 형식적 정의

- **언어 인식기(Recognizers)**
  - 언어의 알파벳으로 구성된 입력 문자열을 읽어서 이 언어에 속하는지를 판단하는 인식 장치

    (예) 컴파일러의 구문분석기

- **언어 생성기(Generators)**
  - 언어의 문장들을 생성하는 장치
  - 특정 문장의 구문이 옳다면 주어진 문법에 의해 언어가 유도된다.

언어의 형식적 정의(continued)

- **언어 인식기(Recognizers)**

  (예) 언어(L)= 0(10)*

  입력 알파벳 = {0, 1}

  입력 문자열 = 01010

  여기서, 01010은 0(10)* 에 속하는지 판단

  ※ 언어(L)에 대한 automata를 그려서 accept/reject 여부를 판단

언어의 형식적 정의(continued)

- **언어 생성기(Generators)**
  - 특정 문장의 구문이 옳다면 주어진 문법에 의해 언어가 유도된다.

(예) 다음 문법은 어떤 언어를 생성하는가?

$S \rightarrow S10 \mid 0$

[풀이]

$S \Rightarrow 0$

혹은 $S \Rightarrow S10 \Rightarrow S1010 \Rightarrow S1010 \ldots 10$

$\Rightarrow 01010 \ldots 10 \Rightarrow 0(10)^*$

따라서 $L = \{0, 010, 01010, \ldots, 01010 \ldots 10\}$

$= 0(10)^*$

BNF and Context-Free Grammars

- 문맥자유문법(Context-Free Grammars, CFG)
  - mid-1950s, by Noam Chomsky
  - 자연어의 구문을 기술하기 위한 용도
  - 언어생성기에 활용
  - 프로그래밍언어의 구문 기술에 적합

- Backus-Naur Form (BNF, 1959)
  - Algol 58의 구문 정의를 위해, by John Backus
  - BNF는 문맥자유문법과 동일

BNF 기본원리

- 구문구조를 표현하기 위해 추상화(abstraction)를 사용
- 논터미널 심볼 혹은 터미널로 구성

(ex) 추상화 <assign>의 정의

<assign> → <var> = <expression>

<var> → a | b | c | d

- 터미널은 어휘항목이나 토큰

- Left-hand side (LHS): 논터미널
- Right-hand side (RHS): a string of terminals and/or nonterminals

BNF 기본원리(continued)

- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

```
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
```

- Grammar: a finite non-empty set of rules
- A *start symbol* is a special element of the nonterminals of a grammar

BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

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

Describing Lists

- Syntactic lists are described using recursion

```
<ident_list> → ident
             | ident, <ident_list>
```

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

An Example Grammar

<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → a | b | c | d

<expr> → <term> + <term> | <term> - <term>

<term> → <var> | const

An Example Derivation

```
<program> => <stmts> => <stmt>
                     => <var> = <expr>
                     => a = <expr>
                     => a = <term> + <term>
                     => a = <var> + <term>
                     => a = b + <term>
                     => a = b + const
```

```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```

Derivations

- Every string of symbols in a derivation is a *sentential form*
- A *sentence* is a sentential form that has only terminal symbols
- A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

유도과정 연습(slide17~21)
[문법 예1]

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

"begin A = B + C; B = C end" 문장이 [문법 예1]로 부터 생성되는가?

유도과정 연습 : 유도과정

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

```
<program> => begin <stmt_list> end
          => begin <stmt>; <stmt_list> end
          => begin <var>=<expr>; <stmt_list> end
          => begin A=<expr>; <stmt_list> end
          => begin A=<var>+<var>; <stmt_list> end
          => begin A=B+<var>; <stmt_list> end
          => begin A=B+C; <stmt_list> end
          => begin A=B+C; <stmt> end
          => begin A=B+C; <var>=<expr> end
          => begin A=B+C; B=<expr> end
          => begin A=B+C; B=<var> end
          => begin A=B+C; B=C end
```

[문법 예2(단순 배정문)]

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

유도과정

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

- A = B * (A + C)의 최좌단 유도

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

(연습) 최우단 유도 해보기

최우단 유도 해보기

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

- A = B * (A + C)

  <assign> => <id> = <expr>

Parse Tree

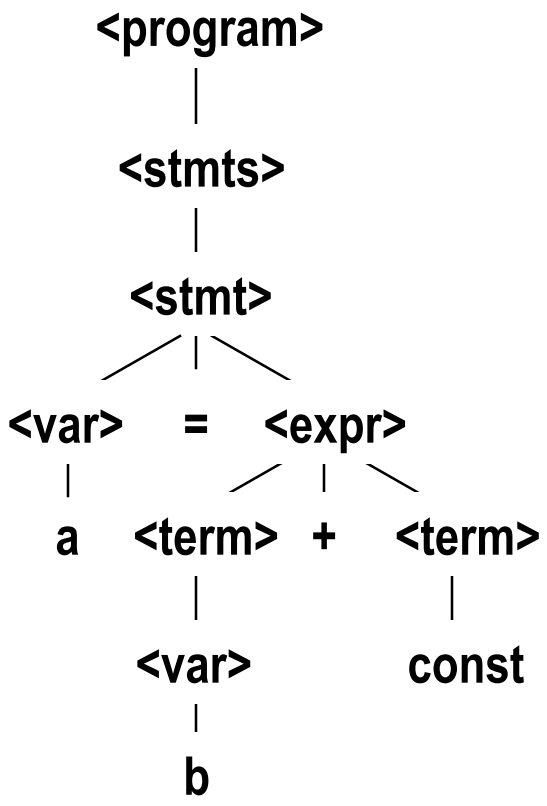
- A hierarchical representation of a derivation

- Top-down parsing: leftmost derivation 순서와 동일
- Bottom-up parsing: rightmost derivation의 역순

Parse Tree: top-down parsing

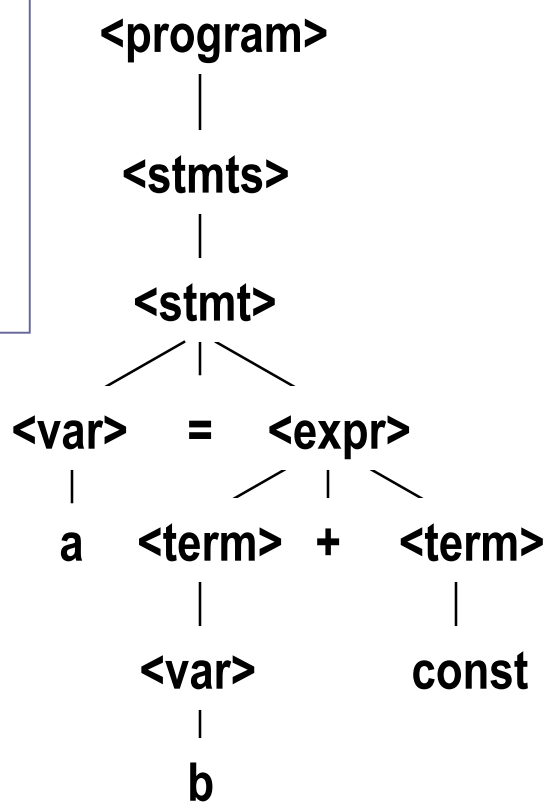
- 내부노드 : 논터미널
- 잎 노드 : 터미널

```
<program> => <stmts>
   => <stmt>
   => <var> = <expr>
   => a = <expr>
   => a = <term> + <term>
   => a = <var> + <term>
   => a = b + <term>
   => a = b + const
```



Parse Tree: bottom-up parsing

```
<program> => <stmts>
 => <stmt>
 => <var> = <expr>
 => <var> = <term> + <term>
 => <var> = <term> + const
 => <var> = <var> + const
 => <var> = b + const
 => a = b + const
```



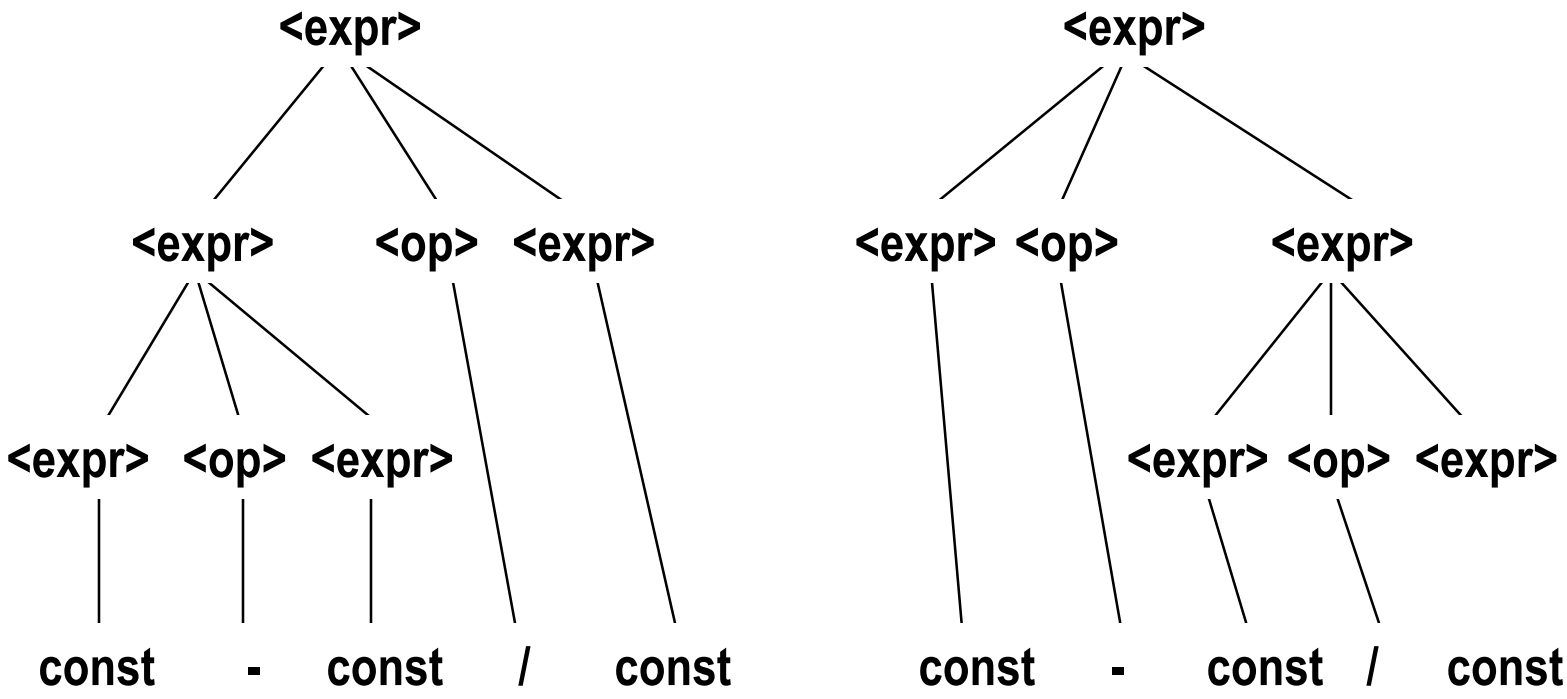
Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const

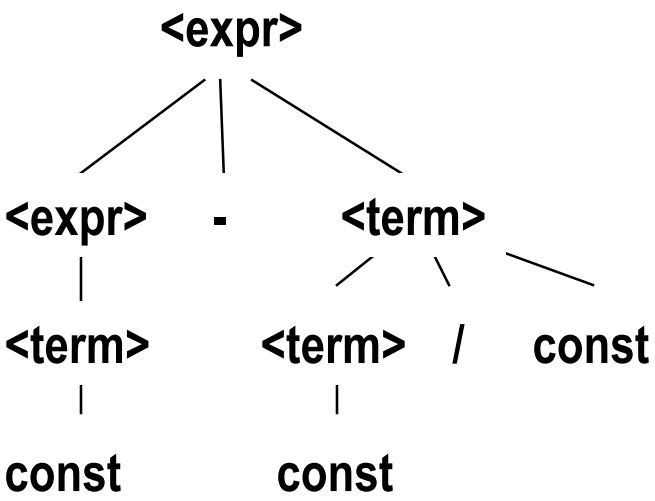
<op> → / | -



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```



모호한 문법 연습(slide 28~32)

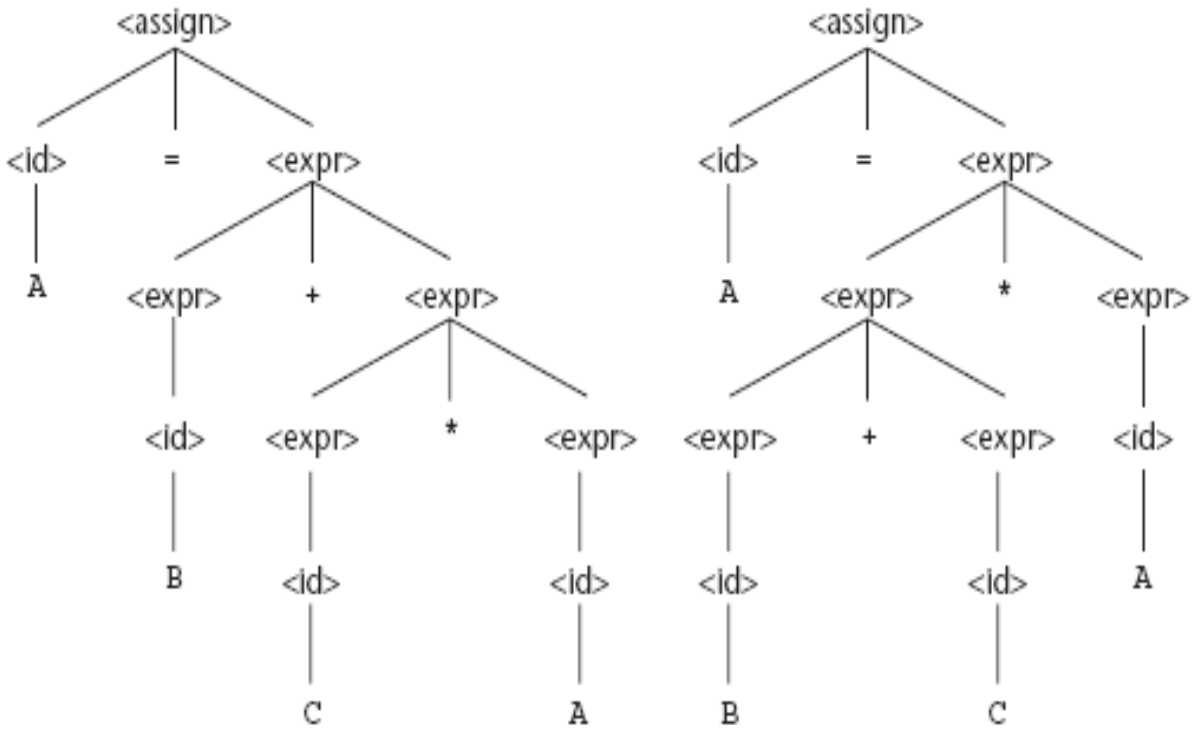
(예) 모호한 연산식 문법

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

모호한 문법 연습

Figure 3.2

Two distinct parse trees for the same sentence,
`A = B + C * A`




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

모호하지 않은 연산식 문법으로 수정

- 모호하지 않으면서도 연산자들의 배치순서와 관계없이 연산자들의 우선순위를 명세하는 문법 필요
  - 파스트리의 낮은 위치의 연산자가 높은 위치의 연산자보다 높은 우선순위를 갖는다.
  - 파스트리 상에 연산자 우선순위가 표현되게 하는 문법을 추가하여 모호성을 제거한다.

→ 별도의 논터미널을 사용하여 다른 우선순위를 갖는 피연산자를 표현하자!!

모호하지 않은 연산식 문법

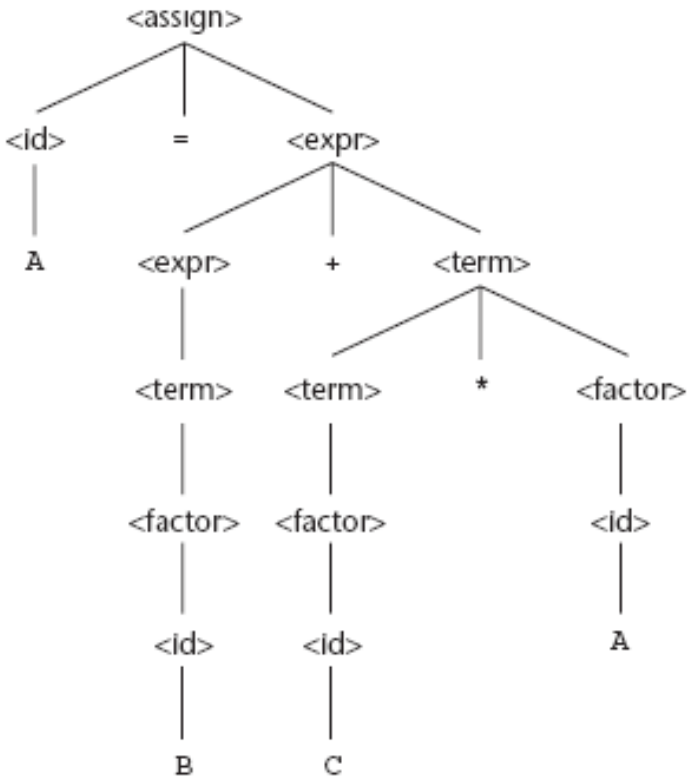
```
<assign> → <id> = <expr>
   <id> → A | B | C
   <expr> → <expr> + <expr>
            | <expr> * <expr>
            | ( <expr> )
            | <id>
```

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

모호하지 않은 연산식 문법(파스 트리)

Figure 3.3

The unique parse tree for A = B + C * A using an unambiguous grammar

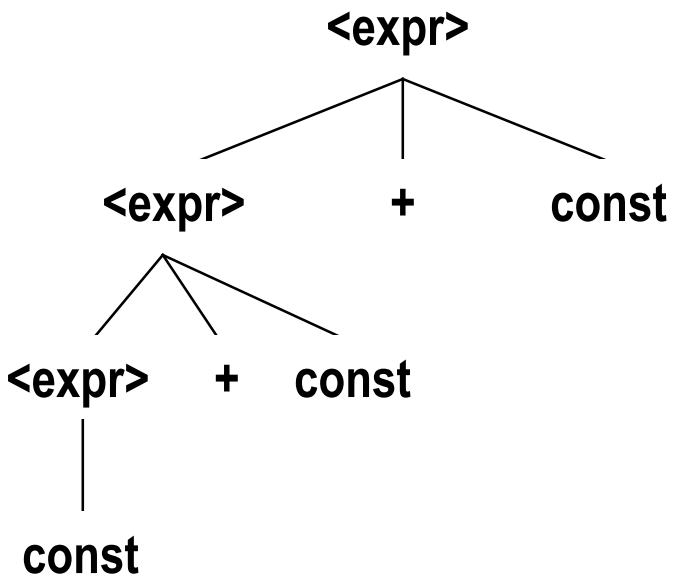



```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
         | <term>
<term> → <term> * <factor>
         | <factor>
<factor> → ( <expr> )
           | <id>
```

연산자의 결합성(Associativity)

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```

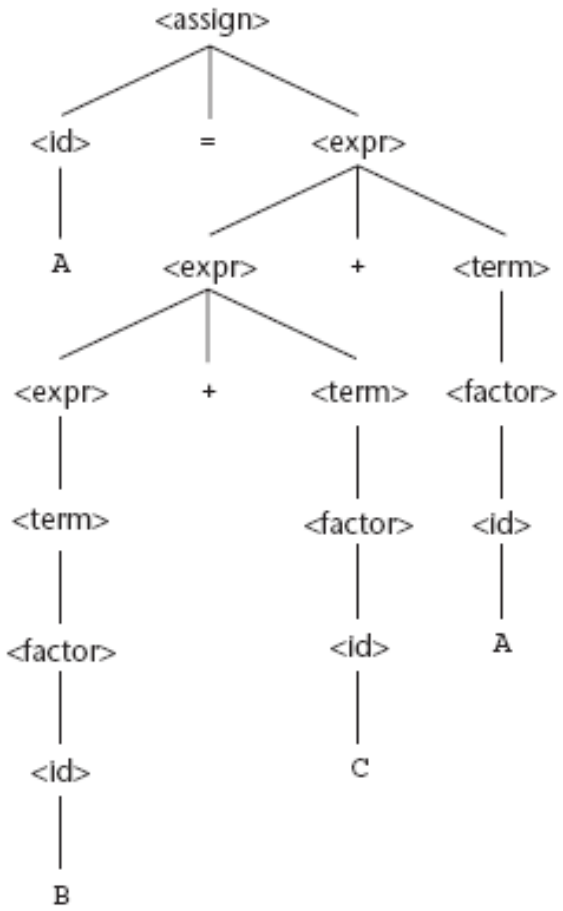


연산자의 결합성(continued)

- A=B+C+A

Figure 3.4

A parse tree for A = B + C + A illustrating the associativity of addition




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

연산자의 결합성 (continued)

- 연산자 결합규칙도 문법상에 표현 가능
  - 규칙에서 LHS가 RHS의 처음에 오면
    좌순환적(left recursive)이라 함
  - 규칙에서 LHS가 RHS의 맨 끝에 오면
    우순환적(right recursive)이라 함
  - 좌순환규칙은 좌결합 법칙을 명세하고, 우순환규칙은 우결합법칙을 명세함

연산자의 결합성 (continued)

다음 문법에서 각 연산자의 결합규칙은?

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

(예) A ** B ** C에 대한 파스트리는?

연산자의 결합성 (continued)

(예) A ** B ** C에 대한 파스트리는?

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

Extended BNF

- Optional parts are placed in brackets [ ]

  `<proc_call> -> ident [(<expr_list>)]`

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

  `<term> → <term> (+|-) const`

- Repetitions (0 or more) are placed inside braces { }

  `<ident> → letter {letter|digit}`

BNF → EBNF

- BNF

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

- EBNF로 표현하면

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

Recent Variations in EBNF

- Use of a colon instead of $\rightarrow$
- Alternative RHSs are put on separate lines

  (ex) statement:

  for(expr-1$_{opt}$; expr-2$_{opt}$; expr-3$_{opt}$) statement

  if(expr) statement

  while(expr) statement

  :
- Use of $_{opt}$ for optional parts
- Use of `oneof` for choices

  (ex) operator $\rightarrow$ `one of + * - / < >`

정적 의미론(Static Semantics)

- 문맥자유문법(CFGs)은 프로그래밍 언어의 모든 구문을 기술할 수 없음
  - 타입호환성규칙(부동소숫점수와 정수 변수)
  - 모든 변수는 사용 전에 선언되어야 하는 규칙

- 타입의 제한 사항은 주로 정적 의미론으로 기술

- 프로그래밍 언어의 구문과 정적 의미론 규칙을 기술하기 위해 설계된 문법 ⇒ 속성 문법

속성문법(Attribute Grammars)

- 의미정보를 파스트리 노드에 추가시킨 **CFG**
  - 속성 : 문법기호(터미널, 논터미널 기호)와 연관;
    값 배정 가능
  - 속성계산함수(의미함수) : 문법규칙과 연관;
    속성값의 계산과정 명세
  - 술어함수: 문법규칙과 연관;
    언어의 구문과 정적 의미론 규칙 서술

Attribute Grammars : Definition

- Def: An attribute grammar is a context-free grammar G = (S, N, T, P) with the following additions:
  - For each grammar symbol *x* there is a set *A(x)* of attribute values
  - Each rule has a set of functions that define certain attributes of the nonterminals in the rule
  - Each rule has a (possibly empty) set of predicates to check for attribute consistency

Attribute Grammars: Definition

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule
- Functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$ define *synthesized attributes*
- Functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$, *for* $1 <= j <= n$, define *inherited attributes*
- Initially, there are *intrinsic attributes* on the leaves

Attribute Grammars: An Example

- Syntax

  <assign> → <var> = <expr>

  <expr> → <var> + <var> | <var>

  <var> → A | B | C

- actual_type: synthesized for <var> and <expr>
- expected_type: inherited for <expr>

Attribute Grammar (continued)

- Syntax rule: `<expr> → <var>[1] + <var>[2]`

  Semantic rules:

  `<expr>.actual_type ← <var>[1].actual_type`

  Predicate:

  `<var>[1].actual_type == <var>[2].actual_type`

  `<expr>.expected_type == <expr>.actual_type`

- Syntax rule: `<var> → id`

  Semantic rule:

  `<var>.actual_type ← lookup (<var>.string)`

Attribute Grammars (continued)

- How are attribute values computed?
  - If all attributes were inherited, the tree could be decorated in top-down order.
  - If all attributes were synthesized, the tree could be decorated in bottom-up order.
  - In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

Attribute Grammars (continued)

<expr>.expected_type ← inherited from parent

<var>[1].actual_type ← lookup (A)
<var>[2].actual_type ← lookup (B)
<var>[1].actual_type =? <var>[2].actual_type

<expr>.actual_type ← <var>[1].actual_type
<expr>.actual_type =? <expr>.expected_type

속성문법: Example-1

- 구문규칙

  <proc_def> → procedure <proc_name>[1]
  <proc_body> end <proc_name>[2];

- 술어함수 : <proc_name>[1].string ==
  <proc_name>[2].string

속성문법: Example-2

- 구문 규칙

```
<assign> → <var> = <expr>
<expr> → <var> + <var>
       | <var>
<var> → A | B | C
```

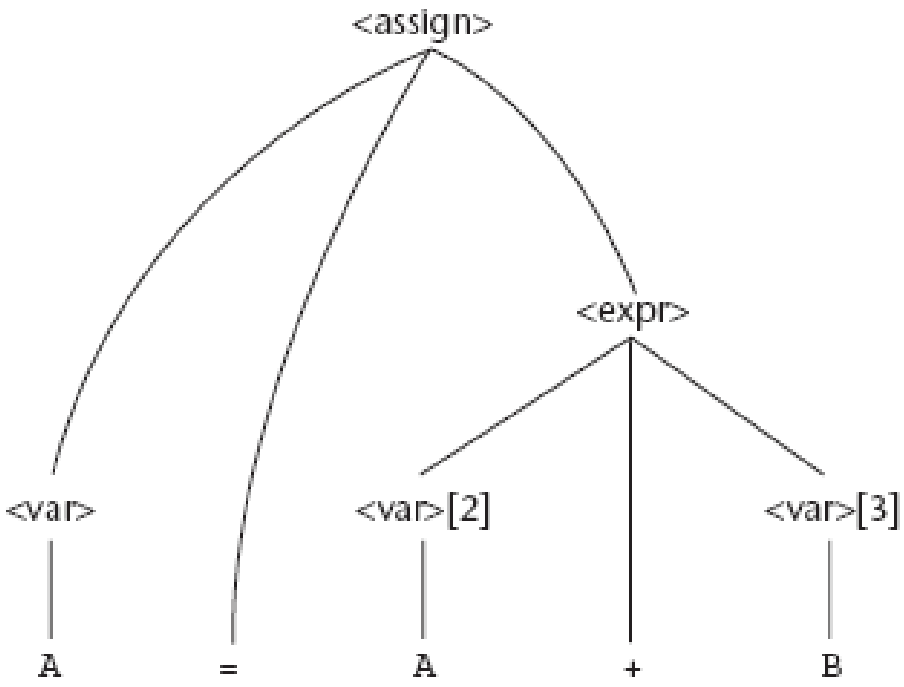
- 논터미널에 대한 속성
  - **actual_type**: 논터미널 <var>과 <expr>에 연관된 합성속성; 변수나 식의 실제타입을 저장
  - **expected_type**: 논터미널 <expr>과 연관된 상속속성; 배정문의 LHS 변수 타입으로 결정

속성문법 (continued)

```
<assign> → <var> = <expr>
<expr> → <var> + <var>
       | <var>
<var> → A | B | C
```

Figure 3.6

A parse tree for
`A = A + B`



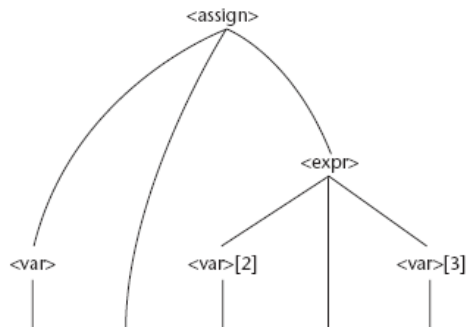
속성문법(continued)

```
<assign> → <var> = <expr>
<expr> → <var> + <var>
       | <var>
<var> → A | B | C
```

1. Syntax rule: `<assign> → <var> = <expr>`
   Semantic rules: `<expr>.expected_type ← <var>.actual_type`
2. Syntax rule: `<expr> → <var>[2] + <var>[3]`
   Semantic rules:
   ```
   <expr>.actual_type ← if(<var>[2].actual_type=int) and
                          (<var>[3].actual_type=int)
                        then int
                        else real   end if
   ```
   Predicate: `<expr>.actual_type == <expr>.expected_type`
3. Syntax rule: `<expr> → <var>`
   Semantic rules: `<expr>.actual_type ← <var>.actual_type`
   Predicate: `<expr>.actual_type == <expr>.expected_type`
4. Syntax rule: `<var> → A | B | C`
   Semantic rule: `<var>.actual_type ← lookup (<var>.string)`

Figure 3.6

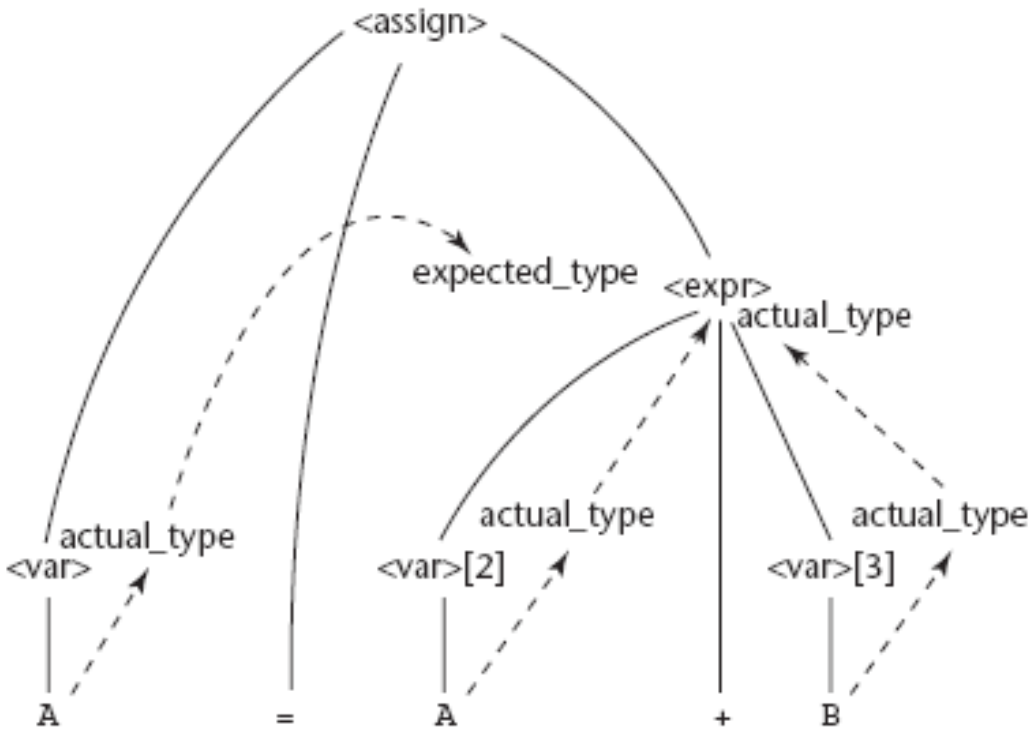
A parse tree for
A = A + B



속성문법 (continued)

Figure 3.7

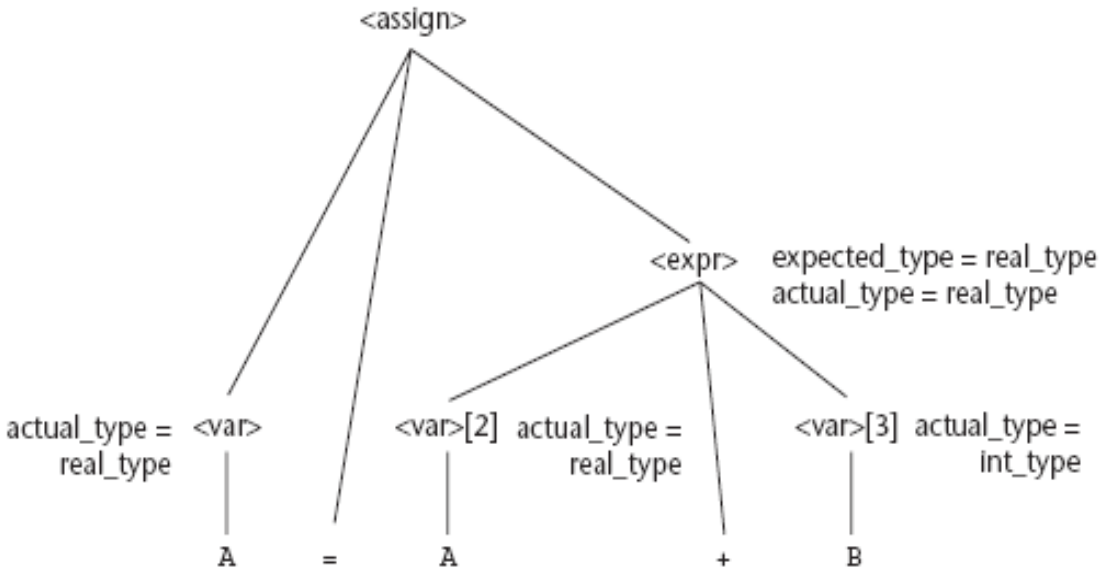
The flow of attributes in the tree



속성문법 (continued)

Figure 3.8

A fully attributed parse tree



변수	타입
A	real
B	int

```
1.  <var>.actual_type ← look-up(A)(규칙4)
2.  <expr>.expected_type ← <var>actual_type (규칙1)
3. <var>[2].actual_type ← lookup(A)(규칙4)
   <var>[3].actual_type ← lookup(B)(규칙4)
4. <expr>.actual_type ← int 또는 real (규칙2)
5. <expr>.expected_type == <expr>.actual_type,
                        이 표현식은 참이나 거짓이다 (규칙3)
```

Semantics

- There is no single widely acceptable notation or formalism for describing semantics
- Several needs for a methodology and notation for semantics:
  - Programmers need to know what statements mean
  - Compiler writers must know exactly what language constructs do
  - Correctness proofs would be possible
  - Compiler generators would be possible
  - Designers could detect ambiguities and inconsistencies

의미론(Semantics)

- 동적 의미론 : 식, 문장, 프로그램 단위의 의미
  - 연산의미론
  - 표기의미론
  - 공리의미론