Syntax and Semantics

국민대학교 컴퓨터공학부

강 승 식

Overview

- Formal Methods of Describing Syntax
  - CFG
  - BNF
  - EBNF
- Derivation, Parse Tree
- Ambiguous/Unambiguous grammar
- Semantics
  - Attribute grammar
  - Operational semantics
  - Denotational semantics
  - Axiomatic semantics
- Example: C-minus language

Syntax of a Language

- Syntax is the set of rules that govern the structure of sentences in a given language

- Sequencing of subject (S), verb (V), and object (O)
  - SVO vs. SOV

Formal language(형식 언어)
Artificial language(인공 언어)

- Language's definition
  - expressions, statements, and program units

- Syntax: form or structure
  - The syntax of "if statement" is …

- Semantics: meaning
  - The meaning of "i = j" is …

Syntax Description

- Context-Free Grammar(CFG): Noam Chomsky
  - Formal grammar
  - Production rule: A → α

    S → NP VP
    NP (정의) → NOUN | PRON
    NOUN → computer | pen | book | …
    PRON → i | you | he | she | it | …
    VP → VERB | VERB NP
    VERB → is | am | go | eat | …

- Backus-Naur Form(BNF): Algol 58 인간으로치면 현대 인간

- Extended Backus-Naur Form(EBNF)

Backus-Naur Form(BNF)

- The same as CFG: num + (num – num) * num

수식 = 수식 operator 수식로 정의

<expression> ::= <expression> <op> <expression>

or <expression> ::= ‘num’ 수식 = num

<expression> ::= ‘(‘ <expression> ‘)’ 수식 = (<수식>)

<op> ::= ‘+’ | ‘-’ | ‘*’ | ‘/’

or or or

<if_stmt> ::= ‘if’ <logic_expr> ‘then’ <stmt>

<logic_expr> ::= …

<stmt> ::= …

CFG or BNF notations

- Nonterminals (nonterminal symbols) 화살표왼쪽부분
- Terminals (terminal symbols) 화살쪽 오른쪽 부분 (nonterminals가 아닌 부분은 다 terminals)
- Start symbol 시작기호
- Production rules 생성규칙들 (->) or ::=
  - LHS → RHS

- Example

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

  statesment 단문 또는 복문

  <single_stmt> → …

  <stmt_list> → …

Example: identifier list

변수명

- <id_list> ::= <id> | <id> , <id_list>
- <id> ::= ‘a’ | ‘b’ | ‘c’

ex) int a,b,c ;

- a
- a, a
- a, b, c
- a, b, b, a

Derivation, Parse Tree

parse, parsing이라는 것은 입력이 먼저 들어와야함

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

문장1 or 문장1; 문장2

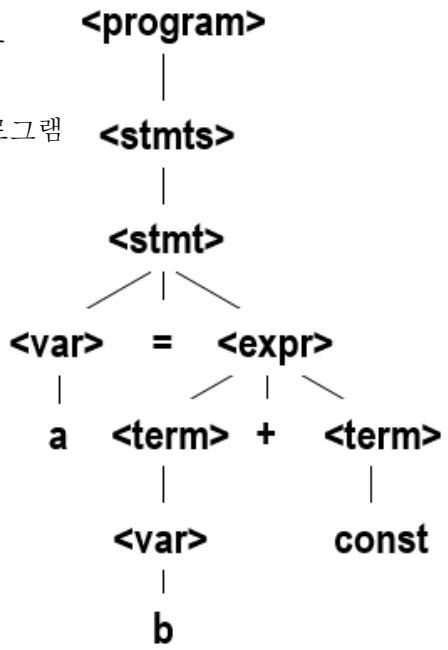
c언어는 문장의 끝마다 ;가 들어가는데 이게 다른점이다.

변수 = 치환문

각각 총 25(a,b,c,d,const 5*5)개 가능

치환문들이 프로그램이 되는 simple한 프로그램

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



변수 + 상수

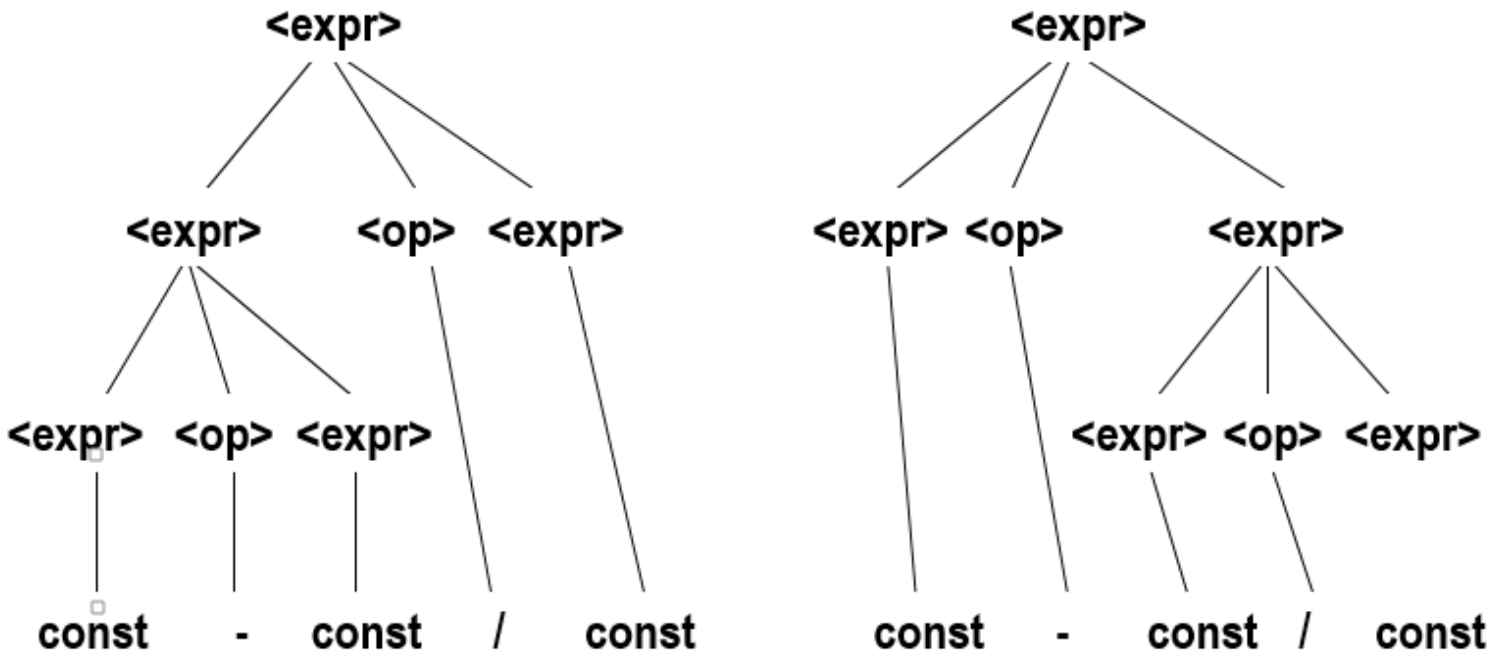
시작기호로부터 생성해 낼 수 있으면 맞는문장 생성못하면 틀린문장

Ambiguous Grammar

parse tree가 두개 이상 나오는 이유는 문법이 잘못되었기 때문
ambiuous grammer. 아래 예는 우선순위가 달라서 계산결과가 다름
/를 먼저 한다고 치면 오른쪽만 정답임

<expr> → <expr> <op> <expr> | const

<op> → / | -

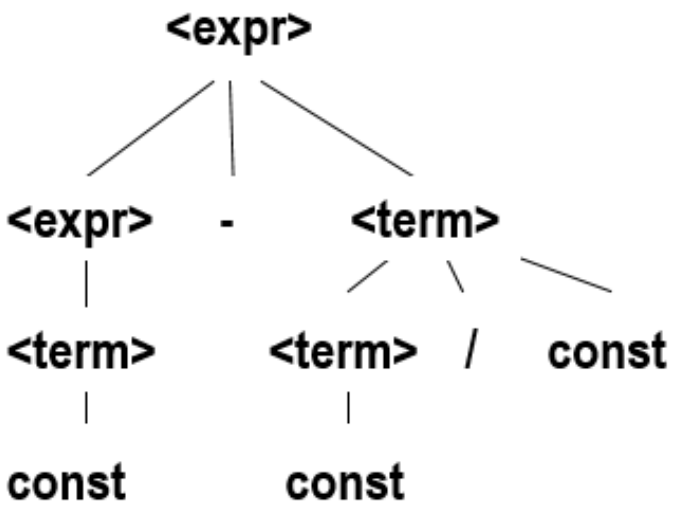


Unambiguous Grammar

- Operator precedence rule

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

시작기호에는 –로만



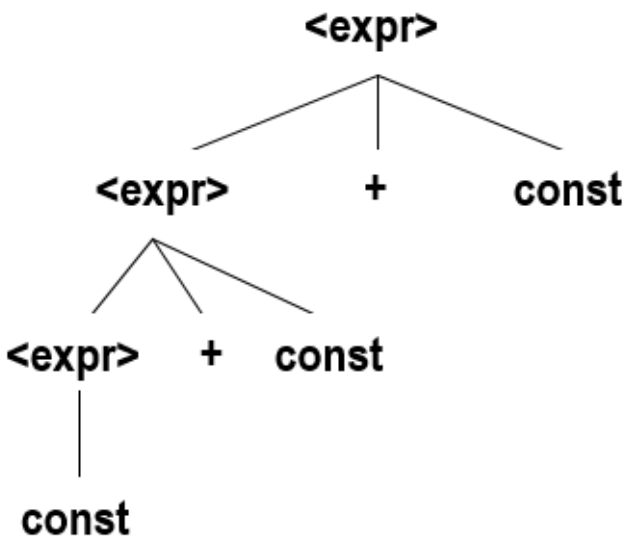
Unambiguous Grammar

- Operator associative rule

연산자가 하나인경우에도 unambiguous 나타남
좌결합을 먼저

하나만 나오면 ambiguous하지 않고 두개이상이 나오면 ambiguous

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



Extended BNF

있어도 되고 없어도 되는

- [ ] --- optional parts

  없어도 되는 부분을 대괄호로

  <if_stmt> ::= ‘if’ <cond> ‘then’ <stmt> [‘else’ <stmt>]

- ( ) --- alternative parts

  <exp> ::= <exp> (‘+’ | ‘-’ | ‘*’ | ‘/’) <exp>

예전 컴파일러들은 변수에 len 정해뒀지만 요즈음 컴파일러들은 변수명 함수명 길이 제약이 없음

- { } --- repetitions

  <id> ::= <letter> { <letter> | <digit> }

  <letter> ::= ‘a’ | ‘b’ | ‘c’ | … | ‘z’

  <digit> ::= ‘0’ | ‘1’ | ‘2’ | … | ‘9’

BNF and EBNF

반복하는부분이 왼쪽에 있으면 우결합
반복하는 부분이 오른쪽에 있으면 좌결합
a+a+a+a+a
반복하는부분 a+
ex 좌결합 : a +a+a+a+a+a
ex 우결합 : a+a+a+a+a+ a

- BNF

<expr> → <expr> '+' <term>
| <expr> '-' <term>
| <term>
<term> → <term> '*' <factor>
| <term> '/' <factor>
| <factor>

- EBNF

우선순위 낮음
<expr> → <term> {('+' | '-') <term>} 좌결합
<term> → <factor> {('*' | '/') <factor>}
우선순위 높음

Semantics

- Variables must be declared before they are used.
- Type Correspondence in “A = B”

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

- Name Correspondence

```
procedure ‘name’ (void)
begin
    …
end ‘name’
```

Semantics

- Static Semantics
  - Attribute Grammar
    - Synthesized, inherited attributes
- Dynamic Semantics
  - Operational Semantics
    - Computer simulation of execution
  - Denotational Semantics
    - Abstract semantics description by recursive function
  - Axiomatic Semantics
    - Formal program verification by formal logic(predicate calculus)

Attribute Grammar: Example

- Syntax rule:

  `<expr>` → `<var>[1] + <var>[2]`

- Semantic rule:

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

- Predicate:

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

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

Example for S-attributed grammar

- CFG

```
Expr → Expr + Term
Expr → Term
Term → Term * Factor
Term → Factor
Factor → "(" Expr ")"
Factor → integer
```

- Synthesized attributes

```
Expr1 → Expr2 + Term [ Expr1.value = Expr2.value + Term.value ]
Expr → Term [ Expr.value = Term.value ]
Term1 → Term2 * Factor [ Term1.value = Term2.value * Factor.value ]
Term → Factor [ Term.value = Factor.value ]
Factor → "(" Expr ")" [ Factor.value =  Expr.value ]
Factor → integer [ Factor.value = strToInt(integer.str) ]
```

BNF Grammar for C-Minus

1. *program* → *declaration-list*
2. *declaration-list* → *declaration-list declaration* | *declaration*
3. *declaration* → *var-declaration* | *fun-declaration*
4. *var-declaration* → *type-specifier* **ID** **;** | *type-specifier* **ID** **[** **NUM** **]** **;**
5. *type-specifier* → **int** | **void**
6. *fun-declaration* → *type-specifier* **ID** **(** *params* **)** *compound-stmt*
7. *params* → *param-list* | **void** terminal (bold체)
8. *param-list* → *param-list* **,** *param* | *param*
9. *param* → *type-specifier* **ID** | *type-specifier* **ID** **[** **]**
10. *compound-stmt* → **{** *local-declarations statement-list* **}**
11. *local-declarations* → *local-declarations var-declarations* | *empty*
12. *statement-list* → *statement-list statement* | *empty*
13. *statement* → *expression-stmt* | *compound-stmt* | *selection-stmt* | *iteration-stmt* | *return-stmt*
14. *expression-stmt* → *expression* **;** | **;**
15. *selection-stmt* → **if** **(** *expression* **)** *statement* | **if** **(** *expression* **)** *statement* **else** *statement*
16. *iteration-stmt* → **while** **(** *expression* **)** *statement*
17. *return-stmt* → **return** **;** | **return** *expression* **;**
18. *expression* → *var* **=** *expression* | *simple-expression*
19. *var* → **ID** | **ID** **[** *expression* **]**
20. *simple-expression* → *additive-expression relop additive-expression* | *additive-expression*

21. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**
22. *additive-expression* → *additive-expression addop term* | *term*
23. *addop* → **+** | **-**
24. *term* → *term mulop factor* | *factor*
25. *mulop* → ***** | **/**
26. *factor* → **(** *expression* **)** | *var* | *call* | **NUM**
27. *call* → **ID** **(** *args* **)**
28. *args* → *arg-list* | *empty*
29. *arg-list* → *arg-list* **,** *expression* | *expression*

Keywords: **else if int return void while**

Special symbols: **+ - * / < <= > >= == != = ; , ( ) [ ] { } /* */**

ID = letter letter*
NUM = digit digit*
letter = a | .. | z | A | .. | Z
digit = 0 | .. | 9

Comments: **/* ... */**