구문과 의미론 #1

Concepts of Programming Languages

2024년 1학기

한양대학교 인공지능대학원
임덕선

목차

- 서론
- 용어
- **id**의 문법적 표현
- 문법의 활용
- 문법의 중요성이 대두
- 정규문법과 **CFG**
- **BNF**
- 유도(파생)
- 파스트리

서론

❑ 구문(syntax)

- 표현식, 문장, 프로그램 단위의 형태 또는 구조
- 문법(grammar)로 표현

❑ 의미론(semantics)

- 표현식, 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

서론

- **Syntax: the form or structure of the expressions, statements, and program units**
- **Semantics: the meaning of the expressions, statements, and program units**
- **Syntax and semantics provide a language's definition**
  - Users of a language definition
    - Other language designers
    - Implementers
    - Programmers (the users of the language)

The General Problem of Describing Syntax: Terminology

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`)
- A *token* is a category of lexemes (e.g., identifier)

용어 : lexeme, token, id

❑ 어휘 항목(lexeme)

- 토큰을 이루는 문자열

❑ 토큰(token)

- 의미적으로 구분되는 최소 단위
- Ex) index = 2 * count + 17;

❑ id(identifier)

- 문자열(lexeme)이 변수명, 함수명, 클래스명 등을 이룰때
- 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

❑ 유도 (or 파생)(derivation)

<id> => <letter> (<letter>| <digit>) *
=> i (<letter>| <digit>) *
=> in (<letter>| <digit>) *
=> ind (<letter>| <digit>) *
=> inde (<letter>| <digit>) *
=> index (<letter>| <digit>) *

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)됨

❑ 언어 생성기와 인식기간의 관계

- 형식언어와 컴파일러 설계 이론의 연구 성과

❑ 프로그래머가 생성한 문장이 올바른지를 판단하는 방법

- “문법”

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) 01010은 $0(10)^*$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)됨

- 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, …… , 0101010 …. 10}
= $0(10)^*$

문법의 중요성이 대두

❑ 문맥 자유 문법(CFG: context-free grammars)

- 1950 년대 중반, Noam Chomsky에 의해서 개발
- 자연 언어에 대한 구문 기술 목적으로 4가지 유형 언어로 정의 (Chomsky의 Hierarchy, 1956, 1959)
- 프로그래밍 언어 기술에 유용한 2가지 유형
  - 정규 문법(regular grammers) (type3)
  - 문맥 자유 문법(type2)

❑ BNF(BNF : Backus-Naur Form) (1959)

- John Backus가 고안 (Algol 58 기술)
- 이를 Peter Naur가 수정 (Algol 60 기술)
- BNF는 문맥 자유 문법과 동일

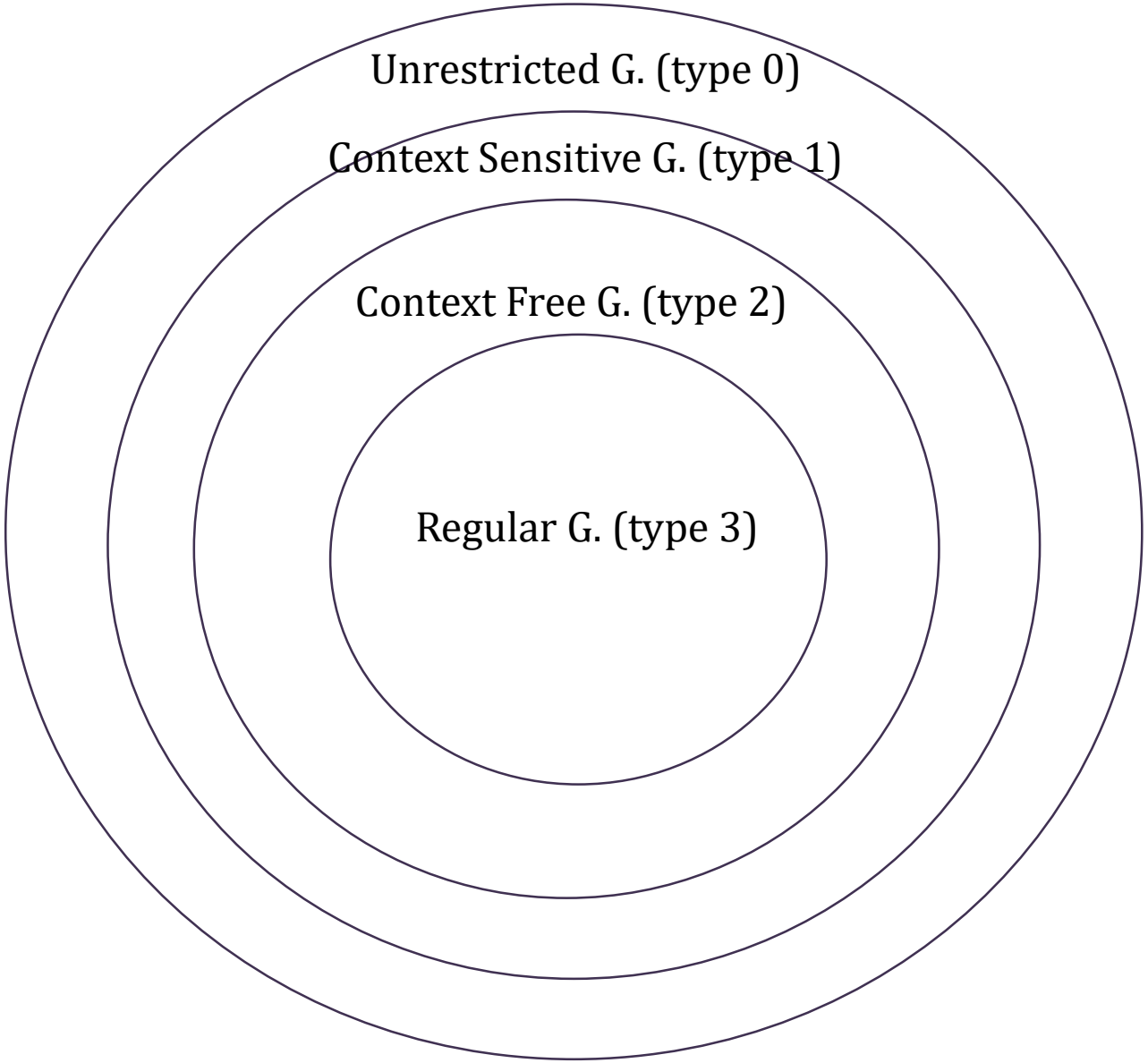
BNF Fundamentals

- In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called *nonterminal symbols,* or just *terminals*)

- *Terminals* are lexemes or tokens

- A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals

정규 문법과 CFG


Unrestricted G. (type 0)
Context Sensitive G. (type 1)
Context Free G. (type 2)
Regular G. (type 3)

❑ Context-Free Grammar (CFG)의 특징

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

❑ **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)

❑ 정규 문법의 특징

A –> βB | β : right-linear G

혹은 **A –> Bβ | β : left-linear G**

여기서, **A**와 **B**는 최대 하나의 **nonterminal,**

β는 **0**개 이상의 **terminal**로 구성 **(**즉**, empty**도 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 생성함

BNF (CFG과 기능상 동일, 표기법만 다름)

❑ 규칙 표현

- LHS, RHS로 구성
  - LHS(left-hand side) : 한 개의 논 터미널(nonterminal symbol)로 표현 (nonterminal을 BNF에서는 각괄호로 나타내고, CFG에서는 대문자로 나타냄)
  - RHS(right-hand side) : 터미널(terminals)과 논터미널들로 구성된 문자열로 표현
  - 터미널은 어휘항목 또는 토큰을 이룸 (terminal을 BNF에서는 각괄호없이 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 생성 규칙들로 구성
- 시작 기호**(start symbols)**은 문법에 포함된 특정 논터미널 기호
- 문법 **G**는 다음 **4**가지로 구성되나, 보통은 **P**로만 나타낸다
- **G = (N, T, S, P)**
  - N : 논터미널들의 집합(a set of nonterminals)
  - T : 터미널들의 집합 (a set of terminals)
  - S ∈ N : 시작 기호(start symbol)
  - P : 생성 규칙들의 집합 (production rules)

BNF 규칙

- 두 개 이상의 **RHS**를 포함 가능

<stmt> –> <single_stmt>
| **begin** <stmt_list> **end**

- 재귀**(recursion)**를 사용하기도 함

<id_list> –> id,
| id, <id_list>

- 명칭리스트의 예**:**

Ex 1) x, y, z
Ex 2) foo1, foo2, index, result

Regular G. 와 CFG

❑ 원래는 다음과 같이 표현해야 함

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

❑ 그러나 보통은 다음과 같이 표현함

<id_list> –> id | id, <id_list>

❑ 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 analyze에서는 <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 **G1**으로부터 생성되는가?

유도 (derivations)

❑ 문장의 생성 과정을 보여주는 일련의 규칙 적용

❑ 방법

1. 시작 기호 **S**부터 시작
2. 문장형태에 논터미널 **X**가 포함되면, **X**를 **LHS**로 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** ->$y_1\ y_2\ \ldots\ y_n$

   S => *$y_1\ y_2\ \ldots\ y_n$ **YZ**

3. 위와 같은 규칙 적용을 반복
4. 문장형태에 논터미널이 포함되지 않으면 유도를 종료
   - 이러한 문장형태를 문장**(sentence)**라 함
   - 문장은 터미널들만 포함

최좌단 (leftmost) 유도와 최우단(rightmost) 유도

❑ 최좌단 유도 **(leftmost derivation)**

– 각 문장 형태에서 가장 좌측에 위치한 논터미널이 규칙 적용을 위해서 선택된다.

❑ 최우단 유도 **(rightmost derivation)**

– 각 문장 형태에서 가장 우측에 위치한 논터미널이 규칙 적용을 위해서 선택된다.

최좌단 (leftmost) 유도와 최우단(rightmost) 유도 (예제)

❑ 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 유도를 보일 것
- 위 문장 생성에 대한 최우단 유도를 보일 것

최좌단 (leftmost) 유도(예제)

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>

유도과정 (최좌단)

<program> => **begin** <stmt_list> **end**
=> **begin** <stmt> ; <stmt_list> **end**
=> **begin** <var> = <expression> ; <stmt_list> **end**
=> **begin** A = <expression> ; <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> = <expression> **end**
=> **begin** A = B + C ; B = <expression> **end**
=> **begin** A = B + C ; B = <var> **end**
=> **begin** A = B + C ; B = C **end**

최우단 (rightmost) 유도(예제)

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>

유도과정 (최우단)

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

문장 유도(예제)

❑ 문법 G2

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

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

최좌단 (leftmost) 유도(예제)

문법 G2

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

유도과정
(최좌단)

< 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)

최우단 (rightmost) 유도(예제)

문법 G2

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

유도과정
(최우단)

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

Q and A