3. 프로그래밍 언어의 명세기법과 해석기법

대구가톨릭대학교

IT공학부

목 차

3.1 프로그래밍 언어의 명세 기법(1)

- **구문(syntax)**
  - 프로그래밍 언어의 구조 의미
  - 프로그램 문장 요소들간의 관계를 나타내는 문장 요소간 결합 법칙
  - 프로그래밍 언어의 구문 표현 -> 일반적으로 BNF 표기법을 사용

3.1 프로그래밍 언어의 명세 기법(2)

- **문맥 자유 문법 (context-free grammar)**
  - 촘스키는 자연 언어에 대해 4가지 언어 유형의 문법을 제안
  - 프로그래밍 언어의 구문을 기술하는데 유용한 것으로 판명
  - 일련의 생성 규칙들로 구성되는데 다음과 같은 형식
    - V → w
- **BNF (Backus-Naur Form)**
  - 프로그래밍 언어의 구문(syntax) 형식을 정의하는 가장 보편적인 표기법
  - 한 언어의 구문에 대한 BNF 정의
    - 언어의 문장을 생성하는 생성 규칙(production rule)들의 집합
  - **생성 규칙(production rule)**
    - 규칙의 왼쪽에는 정의될 대상이 표현되며, 오른쪽에는 그 대상에 대한 정의가 나타남
    - 작성된 프로그램이 구문에 맞는 프로그램인지 인식하기 위한 구문 규율로도 사용

3.1 프로그래밍 언어의 명세 기법(3)

- BNF 표기법에 의한 식별자(identifier) 정의 예

```
<identifier> ::= <letter>| <identifier><letter> | <identifier><digit>
<letter> ::=A | B | C | . . . | X | Y | Z
<digit> ::=0 | 1 | 2 | . . . | 8 | 9
```

- 메타 기호(meta symbol)
  - BNF표기에 사용된 특수기호들(::=, | , <>)
  - 표현하려는 언어의 일부분이 아니고 그 언어를 표현하려고 사용된 특수 기호

meta symbol	의미
::= or →	**정의** : 좌측이 우측으로 대체될 수 있다는 의미
|	**택일**
< > or 대문자	nonterminal : 다시 정의될 심볼
문자열, 예약어 or 소문자	terminal : 더 이상 정의가 불필요한 심볼

3.1 프로그래밍 언어의 명세 기법(4)

<지정문> ::= <변수> := <식>

<식> ::= <식>-<식> | <식>*<식> | (<식>) | <변수> | <수>

<변수> ::= A | B | C

<수> ::= <수><숫자> | <숫자>

<숫자> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

〔그림 3.6〕 간단한 지정문의 문법

<지정문> ⇒ <변수> := <식>

⇒ <변수> := <식> - <식>

⇒ <변수> := <변수> - <식>

⇒ <변수> := <변수> - <수>

⇒ <변수> := <변수> - <수><숫자>

⇒ <변수> := <변수> - <수><숫자><숫자>

⇒ <변수> := <변수> - <숫자><숫자><숫자>

⇒ A := <변수> - <숫자><숫자><숫자>

⇒ A := B - <숫자><숫자><숫자>

⇒ A := B - 1<숫자><숫자>

⇒ A := B - 13<숫자>

⇒ A := B - 135

〔그림 3.7〕 "A:=B-135"의 유도과정

3.1 프로그래밍 언어의 명세 기법(5)

- **EBNF (Extended Backus-Naur Form)**
  - BNF 표기법을 확장하여 보다 읽기 쉽고, 간단하게 표현된 표기법
  - 예) 복합문(compound statement)에 대한 BNF와 EBNF 표현
    - BNF표현

      <compound-statement>::=begin <statement-list>end

      <statement-list>::=<statement>|<statement-list>; <statement>
    - EBNF표현

      <compound-statement>::=begin<statement>{; <statement> } end
  - EBNF는 반복되는 최대횟수와 최소횟수 명시 가능

meta symbol	의미
{ }	0 번 이상 반복
[ ]	0 또는 한 번 나타날 수 있음
(|)	택일 연산자

3.1 프로그래밍 언어의 명세 기법(6)

- 예) if 문장의 EBNF 표현

  <if-statement>::=if <condition> then <statement> **[else <statement>]**

- 예) 사칙연산의 BNF 표현

  <expression>::=<expression> + <expression> |

  <expression> - <expression> |

  <expression> * <expression> |

  <expression> / <expression>

  - 괄호와 택일 연산자를 사용한 사칙연산의 EBNF 표현

  <expression>::= <expression> **(+ | - | * | /)** <expression>

- { }, [ ], |, ( ), ::= 와 같은 메타 기호를 언어의 terminal로 사용하는 경우, '|', '::=' 와 같이 인용부호로 묶어 표현

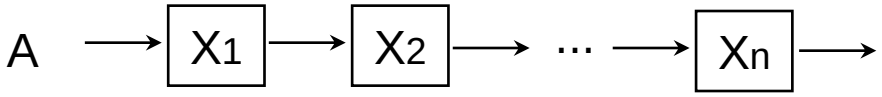
3.1 프로그래밍 언어의 명세 기법(7)

- **구문 도표 (syntax diagram)**
  - EBNF 방법 외에 구문에 대한 형식 정의를 하는 방법
  - 구문을 도식적으로 기술하는 방법
  - 형태가 순서도와 유사
  - 구문 도표는 EBNF 와 일대일 대응
    - 비단말 기호는 사각형으로, 단말기호는 원이나 타원으로, 선과 화살표를 이용해서 순서를 표현
      - terminal x

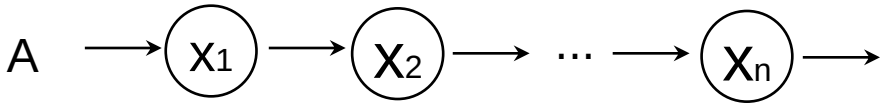
      - nonterminal B

3.1 프로그래밍 언어의 명세 기법(8)

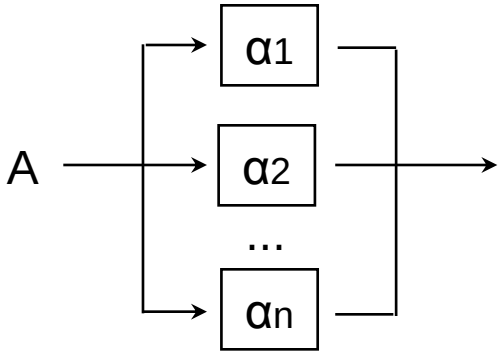
- 구문 도표 그리는 방법
  - A ::= X1X2 ...Xn
    - ① Xi가 nonterminal 인 경우



    - ② Xi가 terminal인 경우



  - A ::= α1 | α2 | … | αn

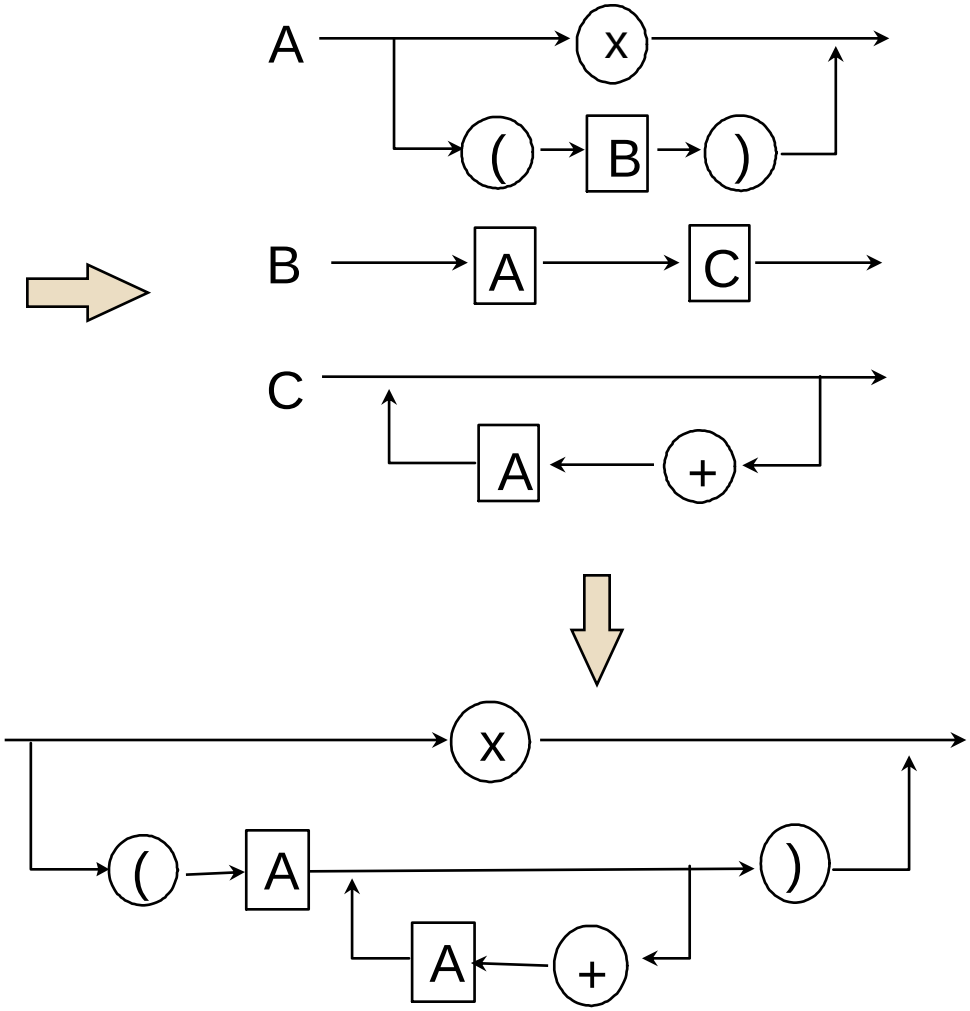


3.1 프로그래밍 언어의 명세 기법(9)

- 예제 2)
  - EBNF ⟶ 구문도표

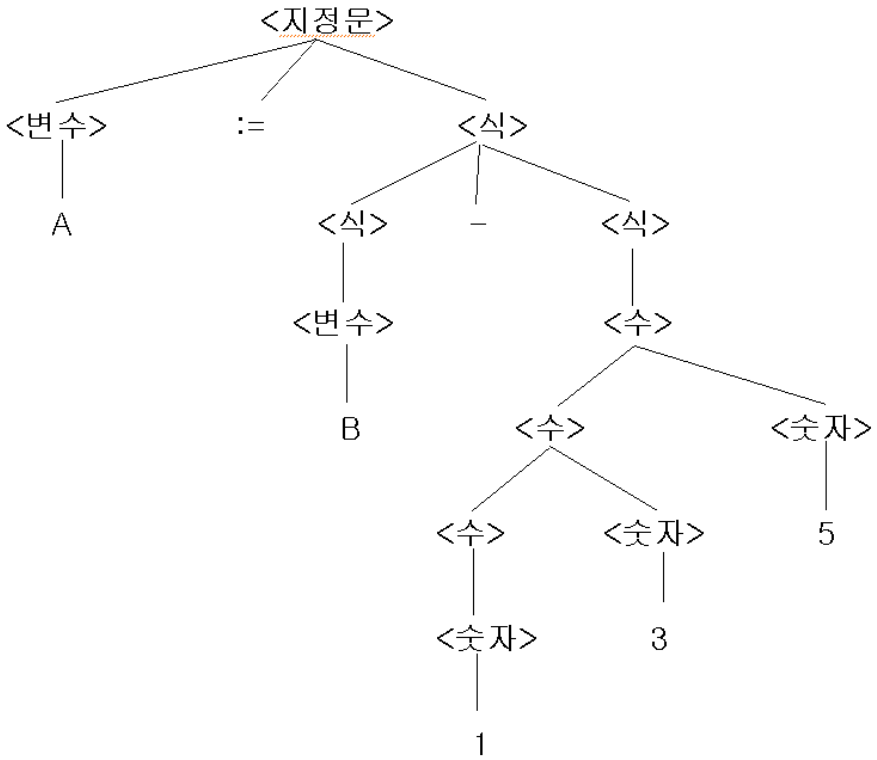
A ::= x | '(' B ')'
B ::= AC
C ::= {+A}

비단말기호 : A, B, C
단말기호 : +, x, (,)



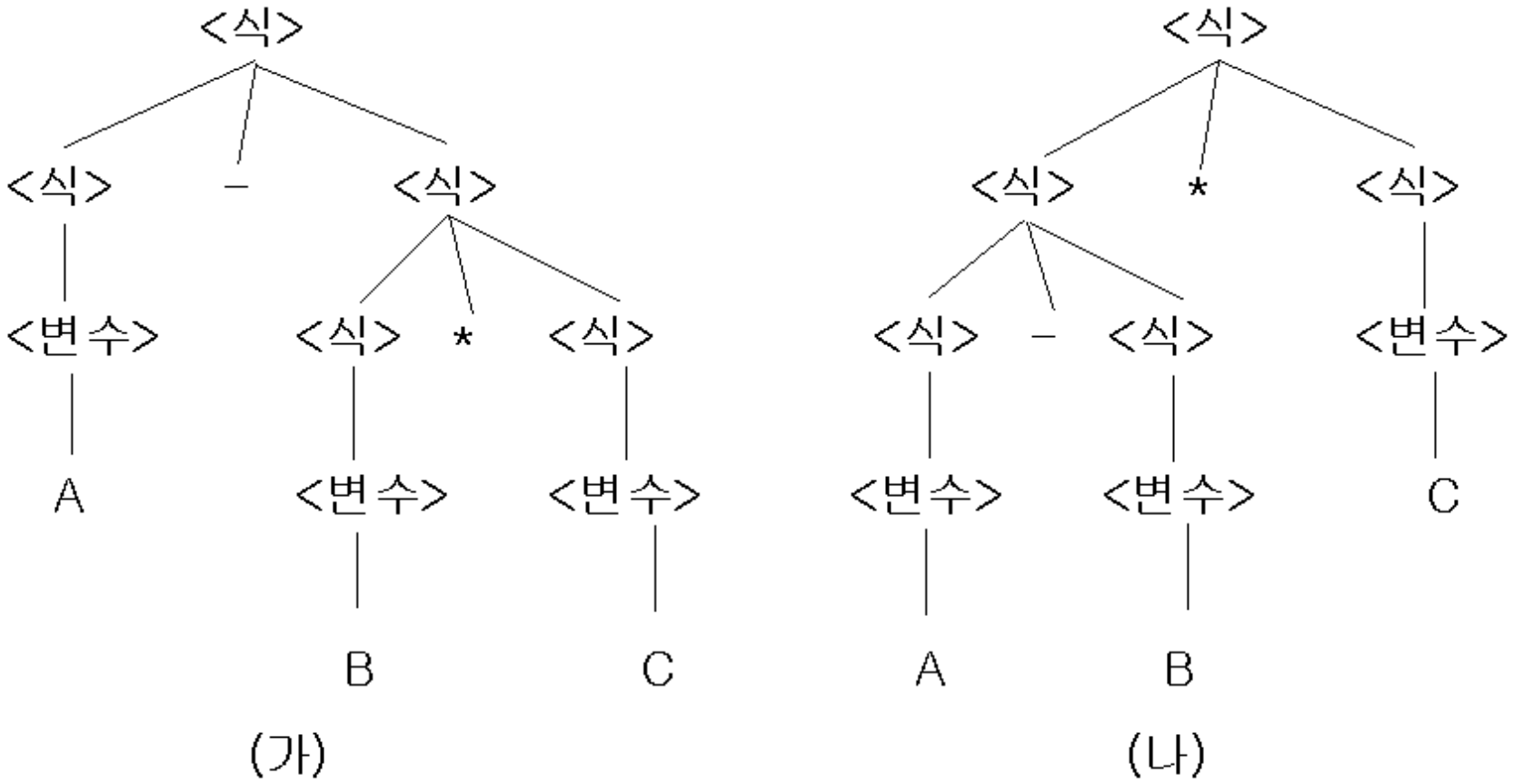
3.1 프로그래밍 언어의 명세 기법(10)

- **파스 트리 (Parse Tree)**
  - 한 표현이 주어진 BNF에 의해 작성될 수 있는지 여부를 나타냄
    - 대상을 root로 하고, 단말노드들을 왼쪽에서 오른쪽으로 나열
    - 주어진 표현에 대한 파스 트리가 존재하면 그 표현은 주어진 BNF에 의해 작성되었다고 말함
  - 예제 1) **A := B - 135** 에 대한 파스 트리



3.1 프로그래밍 언어의 명세 기법(11)

- 예제 2) A-B*C 에 대한 파스 트리



3.1 프로그래밍 언어의 명세 기법(12)

- 동일 문장에 대해서 2개 이상의 서로 다른 파스 트리가 발생하면 이러한 문법을 **모호하다(ambiguous)**고 함
  - 예제 2에서는 1) 연산자 우선순위 모호성, 2) 연산자 결합법칙 모호성이 존재

- 모호한 문법은 문법에서 모호성을 제거하거나 **모호성 제거 규칙 (disambiguating rule)**을 기술해야 함

① 새로운 비단말 기호(예제에서는 <항>)와 문법 규칙을 추가 -> 모호한 문법을 모호하지 않은 문법으로 수정

```
<식> ::= <식>-<식> | <항>
<항> ::= <항>*<항> | <변수> | <수>
```

-> **연산자 우선순위에** 대한 모호성 제거

② 동일한 표현이 좌결합 또는 우결합으로 될 수 있으므로, 문법에 맞는 좌결합 규칙을 명확히 하도록 문법을 고쳐줌

```
<식> ::= <식>-<식> 을
<식> ::= <식>-<항> 으로 대치
```

-> 좌결합으로 파싱,
연산자 결합법칙에 대한 모호성 제거

3.1 프로그래밍 언어의 명세 기법(13)

1) 연산자 우선순위

- 산술식의 한 연산자가 파스 트리의 아랫쪽에 생성된다(먼저 평가되어야 하는)는 것 --> 파스 트리의 윗 쪽에 생성되는 연산자에 비해서 높은 우선순위를 갖는다는 것을 의미
- A-B*C 에 대한 파스 트리
  - (가) 파스트리 : 곱셈 연산자가 뺄셈 연산자보다 높은 우선순위 가짐
  - (나) 파스트리 : 뺄셈 연산자가 곱셈 연산자보다 높은 우선순위 가짐
- 문제발생 이유 : - 연산자와 * 연산자의 우선순위를 같게 했기 때문
- 해결방법

  --> **비단말 기호의 추가와 새로운 규칙을 정의**

우선순위가 높은 *연산자가 -연산자보다 파스트리의 아래쪽에 위치하도록 문법규칙을 개정

<식> ::= <식>-<식> | <항>
<항> ::= <항>*<항> | <변수> | <수>

--> 여전히 문제점 존재 (예: A-B-C)

3.1 프로그래밍 언어의 명세 기법(14)

2) 연산자 결합법칙

- **동일한 우선순위를 갖는 연산자가 두 개 이상 인접해서 나타나는 표현식**에 대해 **좌결합**을 적용할 수도 있고, **우결합**을 적용할 수도 있음 --> 이를 명확히 정의해 줘야 함
  - 예) A-B-C를 (A-B)-C, A-(B-C) 두가지로 해석할 수 있음

- 좌 결합법칙 : 좌순환 규칙(left recursive production)
- 우 결합법칙 : 우순환 규칙(right recursive production)

- A-B-C에 대한 결합법칙 모호성을 제거하려면 → **좌순환 규칙 적용**

<식> ::= <식>-<식> 을
<식> ::= <식>-<항> 으로 대치

3.1 프로그래밍 언어의 명세 기법(15)

- 모호하지 않게 개정한 문법

```
<지정문> ::= <변수> := <식>
<식> ::= <식>-<항> | <항>
<항> ::= <항>*<인수> | <인수>
<인수> ::= (<식>) | <변수> |  <수>
<수> ::= <수><숫자> | <숫자>
<변수> ::= A | B | C
<숫자> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

3.2 프로그래밍 언어 해석 기법들(1)

- **컴퓨터 사용자들은 주로 기계어와 상이한 고급 언어로 프로그램을 작성**
- **고급 언어로 작성된 프로그램을 실제 기계에서 어떻게 실행시킬 것인가 하는 문제에 직면**
  - 해결 방법
    1) **컴파일(compile) 기법**
    2) **인터프리터(interpretation) 기법**
    3) **하이브리드(hybrid) 기법**

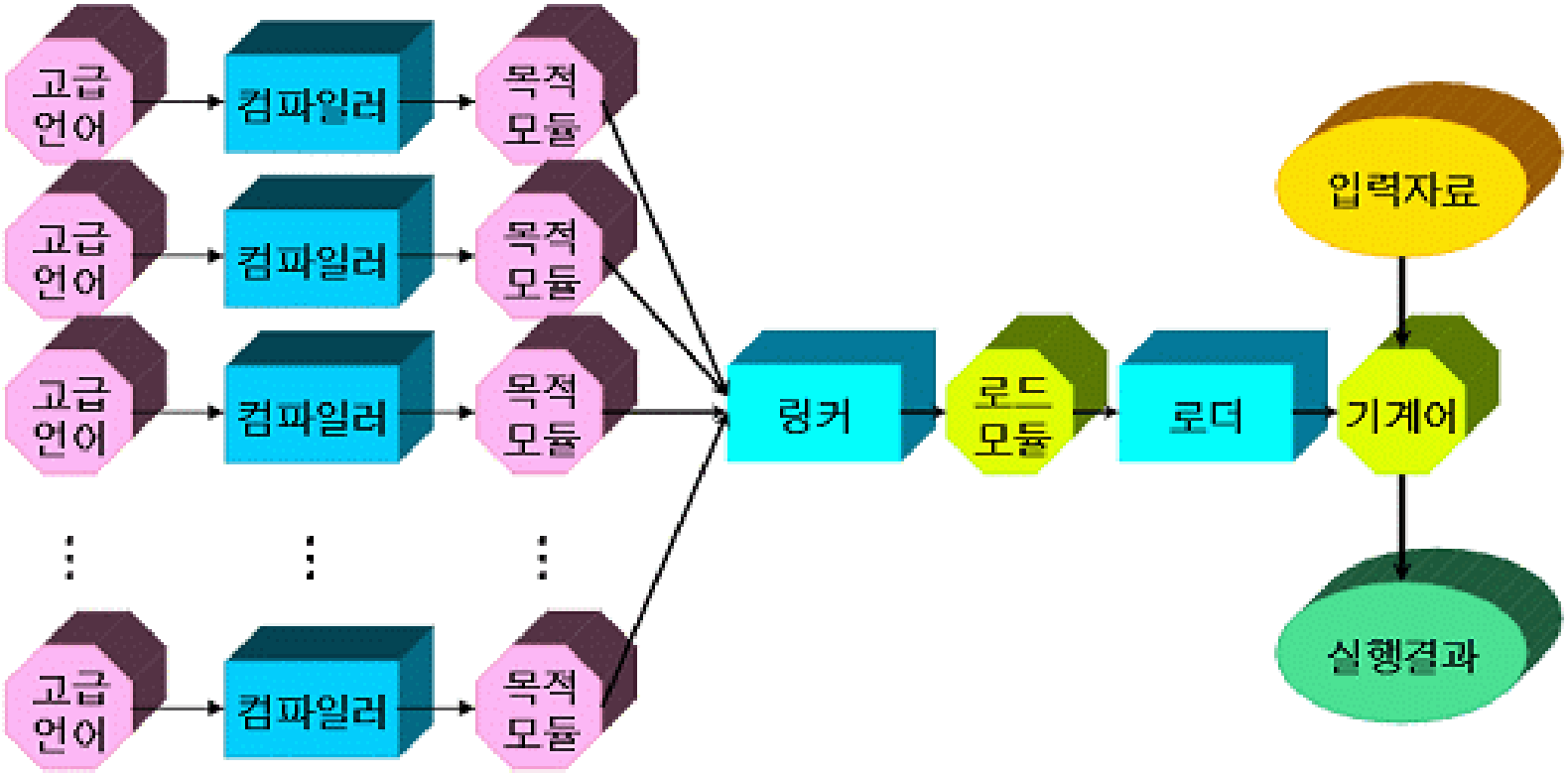
1) **컴파일(compile) 기법**
   - 고급 언어로 작성된 프로그램을 컴퓨터가 바로 실행할 수 있는 프로그램으로 변환하는 방식
   - 컴파일하는 프로그램 : **컴파일러(compiler)**
   - 장점 : 번역이 완료되면 빠르게 프로그램을 실행시킬 수 있음

3.2 프로그래밍 언어 해석 기법들(2)

- **컴파일은 고급 언어로 작성된 프로그램을 기계어로 번역하는 것**
  - 컴파일 된 코드를 목적 코드(object code)라 함
- **번역이 완료된 목적 코드는 그대로 실행할 수 없음**
  - 번역된 목적 코드가 다른 목적 코드나 라이브러리 함수를 호출할 수 있기 때문
- **따라서, 목적 코드에서 호출된 코드를 서로 연결(link)하는 것이 필요**
  - 링크(link) : 목적 코드에 호출되는 코드를 연결하여 하나의 큰 코드를 생성하는 과정
- **목적 코드가 링크(link) 과정을 거치면 실행가능 코드(executable code)가 됨**
- **이 실행가능 코드가 메인 메모리에 적재(load)되면 프로그램이 수행될 수 있음**

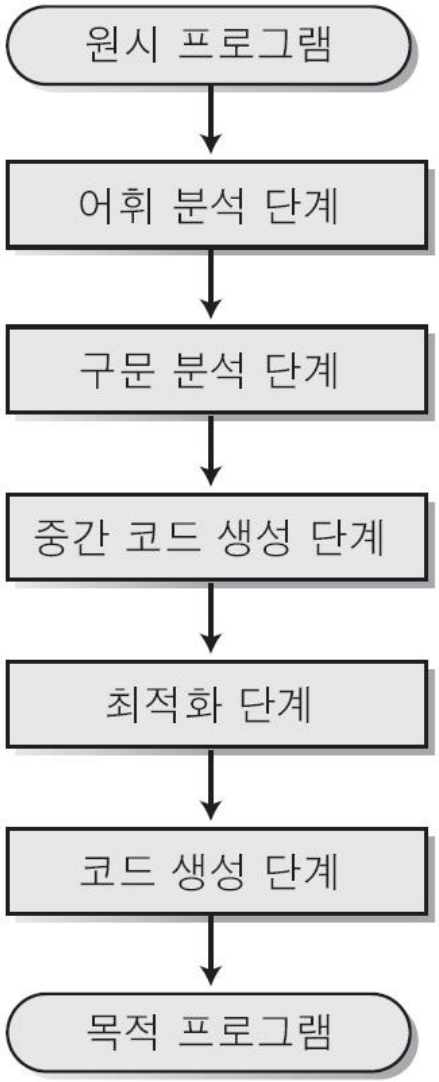
3.2 프로그래밍 언어 해석 기법들(3)

- 프로그램 번역과정



3.2 프로그래밍 언어 해석 기법들(4)

- 컴파일 단계

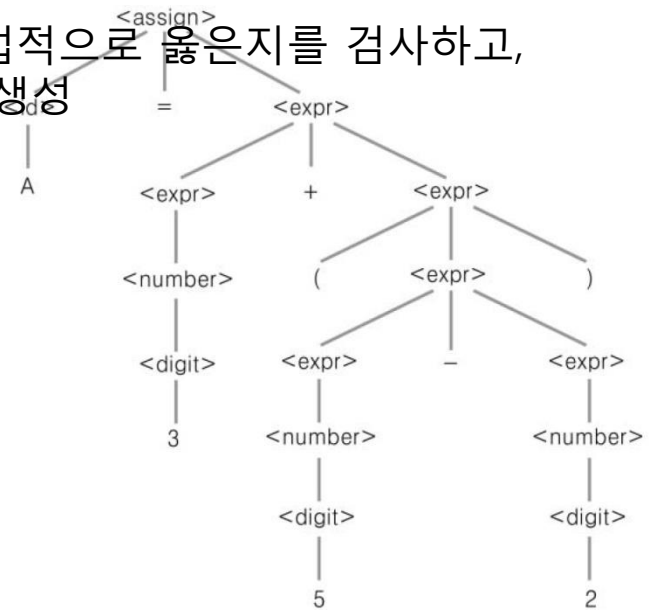


3.2 프로그래밍 언어 해석 기법들(5)

- **어휘 분석 단계**
  - 원시 프로그램을 토큰(식별자, 분리자, 연산자, 숫자, 키워드, 주석 등) 단위 로 자르고, 이러한 토큰과 관련 정보를 구문 분석 단계로 전달
  - 예) A = 3 + (5 – 2)

> 토큰 : A, =, 3, +, (, 5, -, 2,)

- **구문 분석 단계**
  - 어휘 분석 단계에서 전달받은 토큰들이 문법적으로 옳은지를 검사하고, 오류가 없으면 파스 트리라 불리는 구조를 생성
  - 예) A = 3 + (5 - 2)에 대한 파스 트리



3.2 프로그래밍 언어 해석 기법들(6)

- **중간 코드 생성 단계**
  - 기계어는 아니지만 어느 기계에도 의존적이지 않으면서도 기계어에 가까운 중간 코드로 된 프로그램을 생성
  - 이 과정에서 문법적인 오류가 아닌 의미적인 오류를 검사
  - 예) A = 3 + (5 – 2)에 대한 중간 코드

- (sub 5 2 T0) … 'T0 = 5 - 2'를 의미한다.
- (add 3 T0 T1) … 'T1 = 3 + T0'을 의미한다.
- (mov T1 A) … 'A = T1'을 의미한다.

3.2 프로그래밍 언어 해석 기법들(7)

- **최적화 단계**
  - 중간 코드에서 불필요한 코드를 제거하거나 더 효율적인 코드로 개선하여 중간 코드의 크기를 줄이고 실행 속도를 빠르게 함

[예] 중간 코드

(mov 10 A) … 'A = 10'을 의미한다.
(mov A B) … 'B = A'를 의미한다.
(mov B C) … 'C = B'를 의미한다.
(add C 2 D) … 'D = C + 2'를 의미한다.

B와 C를 사용하지 않는 경우, 불필요한 코드를 제거하여 최적화

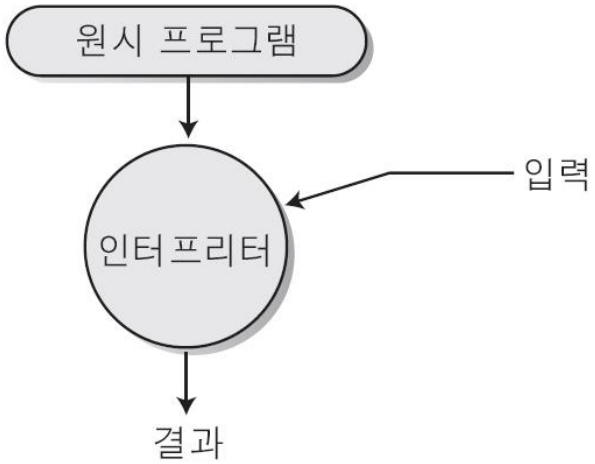
(mov 10 A) … 'A = 10'을 의미한다.
(add A 2 D) … 'D = A + 2'를 의미한다.

- **코드 생성 단계**
  - 최적화된 중간 코드로부터 해당 컴퓨터가 인식할 수 있는 목적 프로그램을 생성

3.2 프로그래밍 언어 해석 기법들(8)

2) 인터프리터 기법(interpretation)

- 고급 언어로 작성된 프로그램을 바로 실행
  - 고급 언어로 작성된 프로그램을 어떠한 번역 과정 없이 바로 실행
- 해석하는 프로그램 : **인터프리터(interpreter)**
- 1960년대 SNOBOL, LISP, Scheme 등이 해석 기법으로 구현되었으나 한동안 사용되지 않다가 JavaScript와 같은 웹 스크립트 언어가 해석 기법으로 구현되고 있음



3.2 프로그래밍 언어 해석 기법들(9)

3) 하이브리드(hybrid) 기법

- 컴파일 기법과 인터프리터 기법을 혼합한 형태
  - 고급 언어로 작성된 프로그램을 해석이 쉽게 되도록 하기 위해 중간 코드 형태로 번역하고 이렇게 번역된 중간 코드 형태의 프로그램을 해석하여 실행
  - 하이브리드 기법의 처리 과정
    - 중간 코드 생성 단계까지는 컴파일 기법으로 동작, 중간 코드를 실행하는 과정은 해석 기법으로 동작
  - 예 : Java