Chapter 2

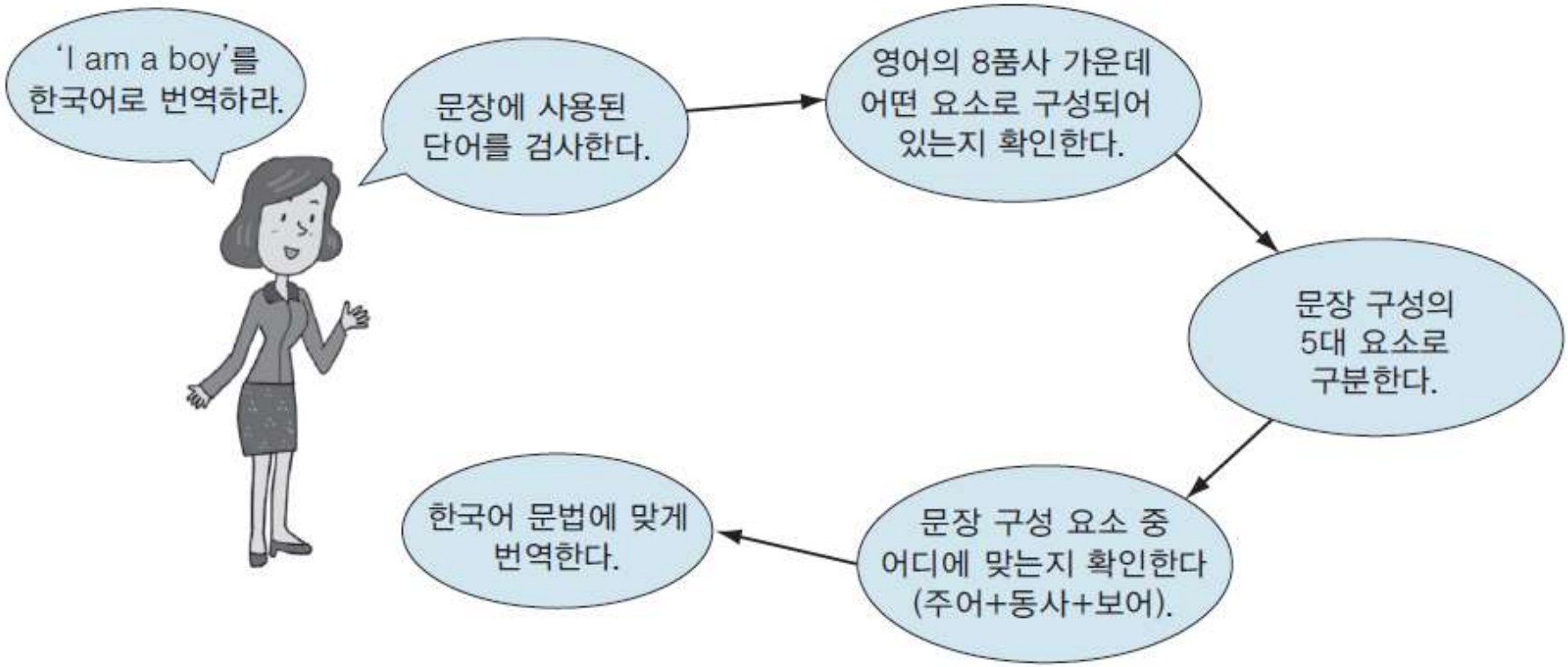
2. 컴파일러 구조 : Part I

수업 목표

- 컴파일 과정에 대한 개념 정립
  - 간단한 프로그래밍 언어에 대해
    - 컴파일 과정 중 전반부를 자세히 들여다 봅니다.
  - 강의자료에서 설명한 내용을
    - 실제 코드로 어떻게 구현했는지 실행시켜 봅니다.

  - **Not see the forest for the trees.**

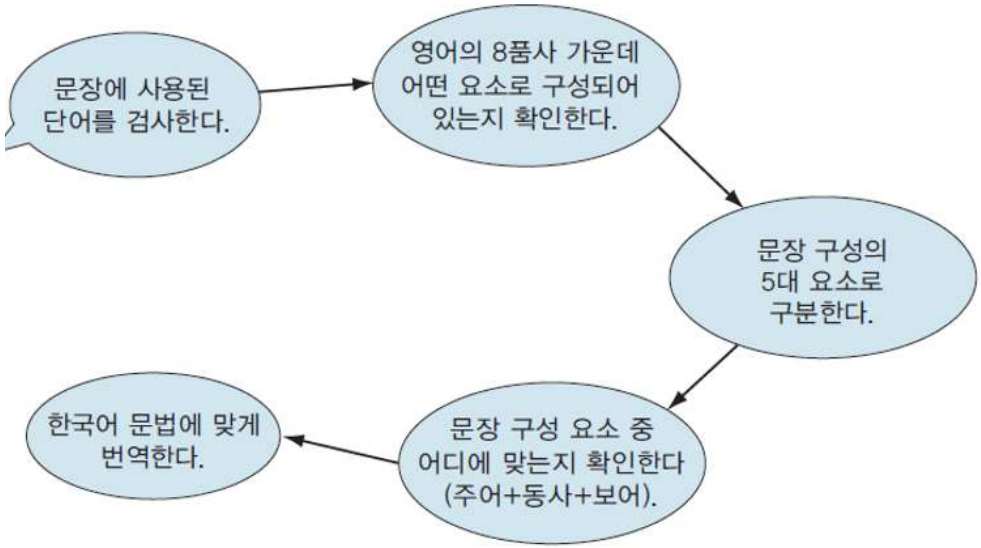
영어 문장을 한글로 번역 **(1/2)**



영어 문장을 한글로 번역 (2/2)

■ 어휘(vocabulary) 분석: 단어 찾기
- **I, am, a, boy, .**

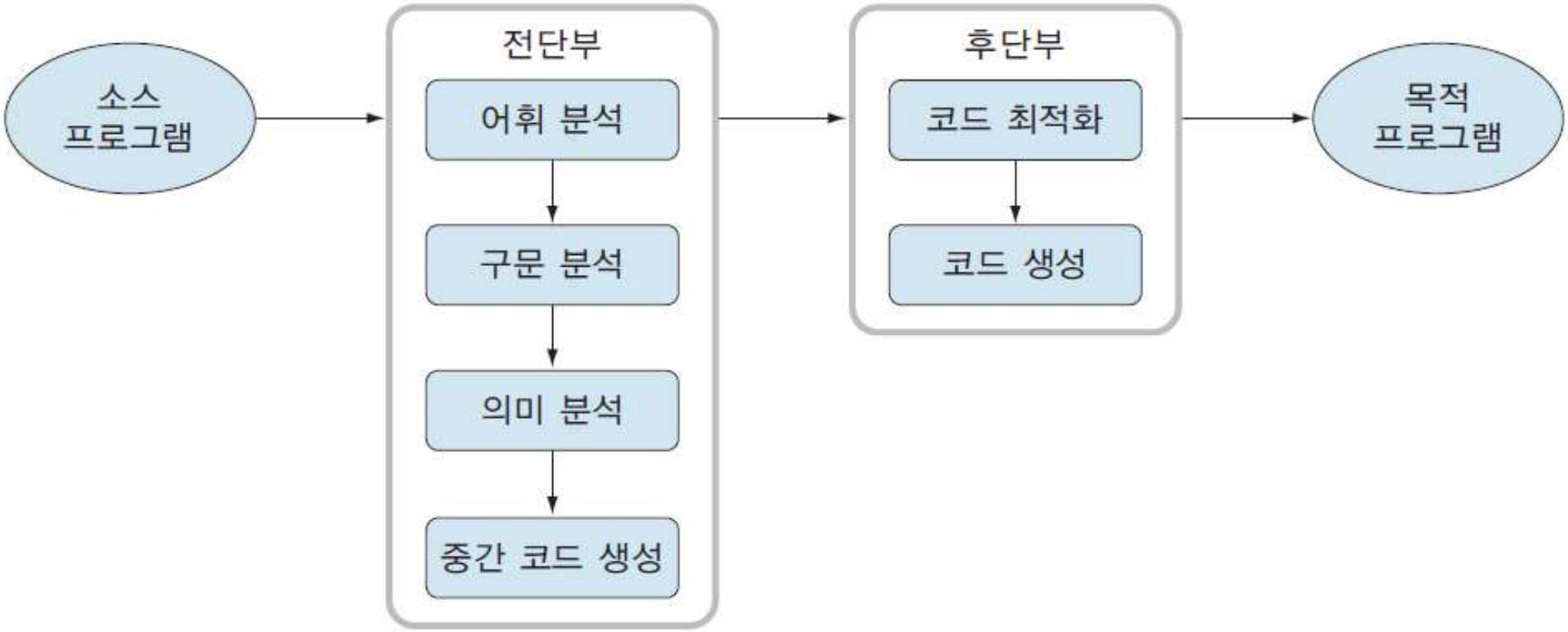
■ 구문 분석
- 5형식 문장 중 해당 문장 형식 확인
  - 단어 품사 확인
  - 2 형식 : **S(I) + V(am) + C(a boy)**



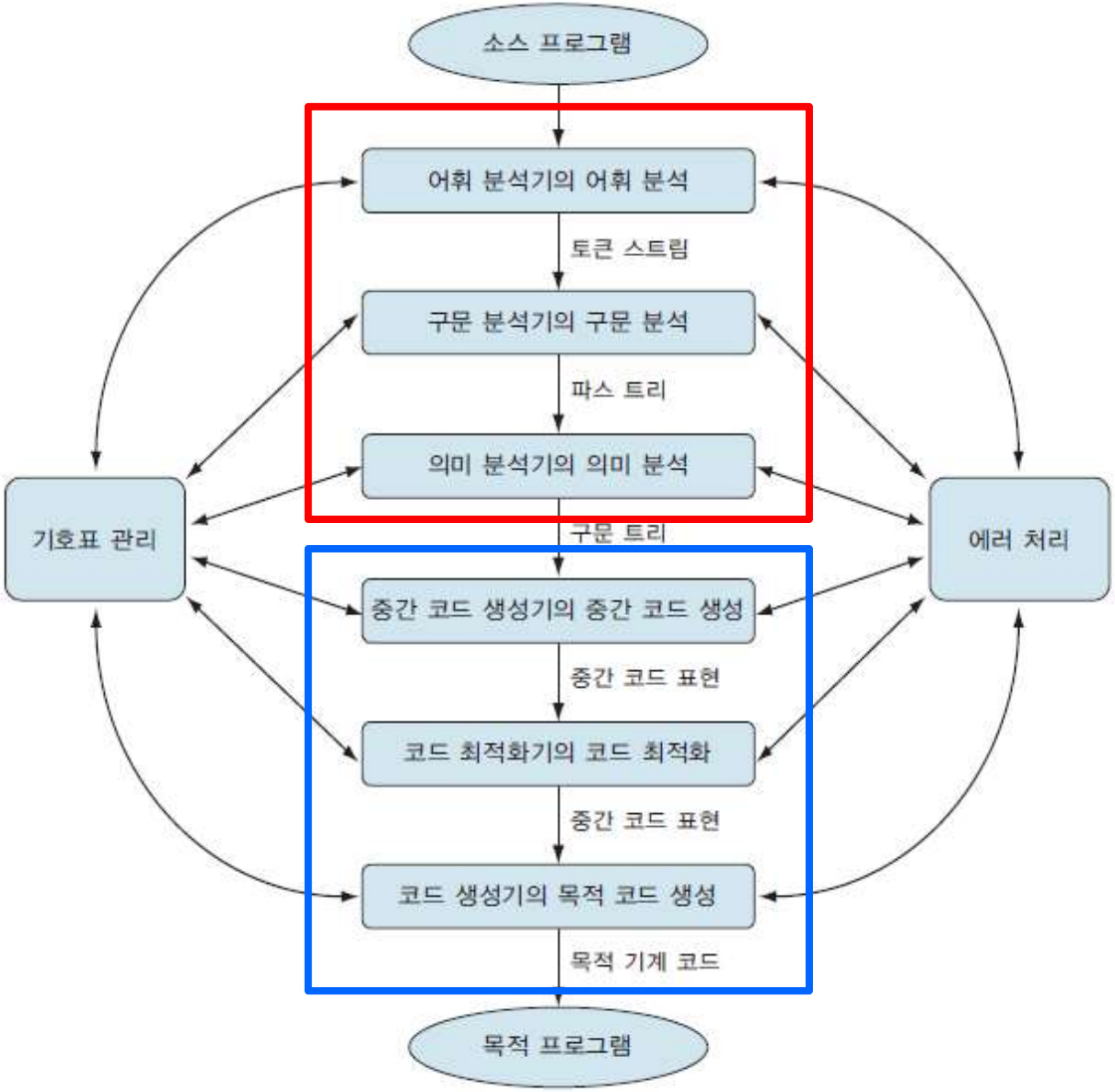
■ 의미 분석
- 1차 번역(초벌 번역) : "나는 한 명의 소년입니다."

■ 코드 최적화
- 번역 완성 : "난 남자야."

컴파일러 구조



컴파일러 상세 구조


소스 프로그램
어휘 분석기의 어휘 분석
토큰 스트림
구문 분석기의 구문 분석
파스 트리
의미 분석기의 의미 분석
구문 트리
중간 코드 생성기의 중간 코드 생성
중간 코드 표현
코드 최적화기의 코드 최적화
중간 코드 표현
코드 생성기의 목적 코드 생성
목적 기계 코드
목적 프로그램
기호표 관리
에러 처리

A very small language : ac

■ ac (adding calculator)

- 데이터 형(types) : 정수, 실수
  - 실수는 소수점 이하 5자리까지만 허용
- 키워드
  - **f** (float), **i** (integer), **p** (print)
- 변수
  - 알파벳 소문자 23자 (키워드 3개 제외)
    - 변수는 사용하기 전에 먼저 선언해야 한다.
- 형 변환
  - 정수 형에서 실수 형 변환은 자동으로 이루어진다.
  - 다른 종류의 형 변환은 허용하지 않는다.

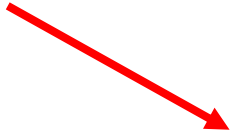
A very small language : Target code

■ dc (desk calculator)

- Stack-based calculator
  - 2 + 3
    - ac 프로그램
  - 2 3 +
    - 코드 생성(reverse polish notation)
  - 5
    - 실행 결과

Context-free grammar (CFG) for ac

15 productions
(생성 규칙)

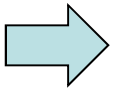


1	Prog	→	Dcls Stmts $
2	Dcls	→	Dcl Dcls
3		|	λ
4	Dcl	→	floatdcl id
5		|	intdcl id
6	Stmts	→	Stmt Stmts
7		|	λ
8	Stmt	→	id assign Val Expr
9		|	print id
10	Expr	→	plus Val Expr
11		|	minus Val Expr
12		|	λ
13	Val	→	id
14		|	inum
15		|	fnum

Context-free grammar for ac.

생성 규칙 (1/2)

Stmt → id assign Val Expr
| print id



Stmt → id assign Val Expr
Stmt → print id

생성 규칙(*Production or Rewriting rule*)

1) 화살표 (→) 왼쪽에 놓인 기호는
화살표 오른쪽 문자열로 확장해서 나타낼 수 있다.

2) | 는 '또는' 이란 뜻.

Stmt 는 **id assign Val Expr** 로 표현할 수 있다.

또는 **Stmt** 는 **print id** 로도 표현할 수 있다.

생성 규칙 (2/2) – 순환 정의, 상세 정의

```
1  Prog  → Dcls Stmts $
2  Dcls  → Dcl Dcls
3        | λ
4  Dcl   → floatdcl id
5        | intdcl id
6  Stmts → Stmt Stmts
7        | λ
8  Stmt  → id assign Val Expr
9        | print id
10 Expr  → plus Val Expr
11       | minus Val Expr
12       | λ
13 Val   → id
14       | inum
15       | fnum
```

반복(순환) 정의

상세 정의

Context-free grammar for ac.

Nonterminal (비단말 기호)

Nonterminals =
{ Prog,
Dcls, Dcl,
Stmts, Stmt,
Expr, Val }

Nonterminal is the symbol on the *left-hand side* (LHS) of productions.

Nonterminal은 생성규칙의 왼쪽, 오른쪽에 모두 사용할 수 있다.

```
 1  Prog  → Dcls Stmts $
 2  Dcls  → Dcl Dcls
 3        | λ
 4  Dcl   → floatdcl id
 5        | intdcl id
 6  Stmts → Stmt Stmts
 7        | λ
 8  Stmt  → id assign Val Expr
 9        | print id
10  Expr  → plus Val Expr
11        | minus Val Expr
12        | λ
13  Val   → id
14        | inum
15        | fnum
```

Context-free grammar for ac.

Start symbol(시작 기호)은 Nonterminal 기호 중 하나

Start symbol

Nonterminals =
{ *Prog*,
Dcls, Dcl,
Stmts, Stmt,
Expr, Val }

단, **Start symbol**은
생성 규칙 왼쪽에
한번만 사용할 수 있다.

```
1  Prog  → Dcls Stmts $
2  Dcls  → Dcl Dcls
3        | λ
4  Dcl   → floatdcl id
5        | intdcl id
6  Stmts → Stmt Stmts
7        | λ
8  Stmt  → id assign Val Expr
9        | print id
10 Expr  → plus Val Expr
11       | minus Val Expr
12       | λ
13 Val   → id
14       | inum
15       | fnum
```

Context-free grammar for ac.

Terminals (단말 기호) (1/4)

terminals =
{ floatdcl, intdcl,
id, assign, print,
plus, minus,
inum, fnum,
$, λ }

They have *no productions*!

Terminal is
the symbol on
the *right-hand side* (RHS)
of productions.

```
1  Prog  → Dcls Stmts $
2  Dcls  → Dcl Dcls
3        | λ
4  Dcl   → floatdcl id
5        | intdcl id
6  Stmts → Stmt Stmts
7        | λ
8  Stmt  → id assign Val Expr
9        | print id
10 Expr  → plus Val Expr
11       | minus Val Expr
12       | λ
13 Val   → id
14       | inum
15       | fnum
```

the end of Input stream

empty string (*null string*)

Context-free grammar for ac.

Terminals (2/4)

terminals =
{ floatdcl, intdcl,
id, assign, print,
plus, minus,
inum, fnum,
$, λ }

inum, fnum , … 이 뭐지**?**

Terminal 기호가
뭔지 어떻게 알 수 있지**?**

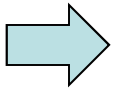
```
1  Prog  → Dcls Stmts $
2  Dcls  → Dcl Dcls
3        | λ
4  Dcl   → floatdcl id
5        | intdcl id
6  Stmts → Stmt Stmts
7        | λ
8  Stmt  → id assign Val Expr
9        | print id
10 Expr  → plus Val Expr
11       | minus Val Expr
12       | λ
13 Val   → id
14       | inum
15       | fnum
```

Context-free grammar for ac.

Terminals (3/4)

terminals =
{ floatdcl, intdcl,
id, assign, print,
plus, minus,
inum, fnum,
$, λ }

- Keywords
  - **f** , **i** , **p**
- Types :
  - 정수(**i**), 실수(**f**)
- Variables
  - 알파벳 소문자 23자
  - reserved words 3개 제외



Terminal	Input symbol
floatdcl	f
intdcl	i
assign	=
plus	+
minus	-
print	p
id	a, b, c,…
inum	12, 345, …
fnum	0.1, 3.14, …

Terminals (4/4) : $와 λ

```
terminals  =
{ floatdcl,  intdcl,
 id,  assign,  print,
 plus,  minus,
 inum,  fnum,
 $,   λ }
```

실제 입력하지 않았지만, 특수한 목적을 위해 사용.

$ (달라): the end of input stream

끝까지 입력을 다 읽었나?

λ (람다) 또는 ε (입실론) : an empty string(or null string)

생략 할 수 있음.

구문 정의를 위해서는

- 생성 규칙이 필요
  - 생성 규칙을 표현하기 위해서는 2종류의 기호가 필요
    - Nonterminal, Terminal
  - Nonterminal 기호는 생성 규칙의 왼쪽, 오른쪽에 모두 사용할 수 있지만,
    - 단, **Start symbol**은 예외
  - Terminal 기호는 생성 규칙의 오른쪽에만 사용할 수 있다.

Check it again!

1 Prog → Dcls Stmts $
2 Dcls → Dcl Dcls
3 | λ
4 Dcl → floatdcl id
5 | intdcl id
6 Stmts → Stmt Stmts
7 | λ
8 Stmt → id assign Val Expr
9 | print id
10 Expr → plus Val Expr
11 | minus Val Expr
12 | λ
13 Val → id
14 | inum
15 | fnum

XML specification

http://www.w3.org/TR/REC-xml/

Python specification

https://docs.python.org/3/reference/grammar.html

ac로 프로그램을 작성해 보자!

input program

```
f  b
i  a
a = 5
b = a + 3.2
p  b
```

문법에 맞게 프로그램을 작성했을까?

```
1  Prog  → Dcls Stmts $
2  Dcls  → Dcl Dcls
3        | λ
4  Dcl   → floatdcl id
5        | intdcl id
6  Stmts → Stmt Stmts
7        | λ
8  Stmt  → id assign Val Expr
9        | print id
10 Expr  → plus Val Expr
11       | minus Val Expr
12       | λ
13 Val   → id
14       | inum
15       | fnum
```

Context-free grammar for ac.

Write a program in ac (1/2)

input program

```
f b
i a
a = 5
b = a + 3.2
p b
```

floatdcl	f
intdcl	i
assign	=
plus	+
minus	-
print	p
id	a, b, c,…

터미널 기호(=토큰)

```
1  Prog  → Dcls Stmts $
2  Dcls  → Dcl Dcls
3        | λ
4  Dcl   → floatdcl id
5        | intdcl id
6  Stmts → Stmt Stmts
7        | λ
8  Stmt  → id assign Val Expr
9        | print id
10 Expr  → plus Val Expr
11       | minus Val Expr
12       | λ
13 Val   → id
14       | inum
15       | fnum
```

ac에 대한 **CFG**

Write a program in ac (2/2)

input program

```
f b
i a
a = 5
b = a + 3.2
p b
```

floatdcl	f
intdcl	i
assign	=
plus	+
minus	-
print	p
id	a, b, c,…

```
1  Prog  → Dcls Stmts $
2  Dcls  → Dcl Dcls
3        | λ
4  Dcl   → floatdcl id
5        | intdcl id
6  Stmts → Stmt Stmts
7        | λ
8  Stmt  → id assign Val Expr
9        | print id
10 Expr  → plus Val Expr
11       | minus Val Expr
12       | λ
13 Val   → id
14       | inum
15       | fnum
```

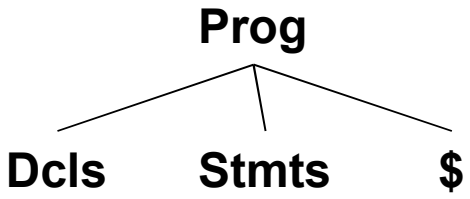
Derivation (유도, 파생) (1/4)

```
 1  Prog  → Dcls Stmts $
 2  Dcls  → Dcl Dcls
 3        | λ
 4  Dcl   → floatdcl id
 5        | intdcl id
 6  Stmts → Stmt Stmts
 7        | λ
 8  Stmt  → id assign Val Expr
 9        | print id
10  Expr  → plus Val Expr
11        | minus Val Expr
12        | λ
13  Val   → id
14        | inum
15        | fnum
```

문장 형태 / 적용한 생성 규칙 번호

Step	Sentential Form	Production Number
1	⟨Prog⟩	
2	⟨Dcls⟩ Stmts $	1

nonterminal <Prog>에 대한 생성 규칙 1번에 따라 <Prog> 가 오른쪽 (right-hand side, RHS) 문자열로 바뀜




```
f  b
i  a
a = 5
b = a + 3.2
p  b
```

Derivation (2/4)

```
 1  Prog  → Dcls Stmts $
 2  Dcls  → Dcl Dcls
 3        | λ
 4  Dcl   → floatdcl id
 5        | intdcl id
 6  Stmts → Stmt Stmts
 7        | λ
 8  Stmt  → id assign Val Expr
 9        | print id
10  Expr  → plus Val Expr
11        | minus Val Expr
12        | λ
13  Val   → id
14        | inum
15        | fnum
```

Step	Sentential Form	Production Number
1	⟨Prog⟩	
2	⟨Dcls⟩ Stmts $	1

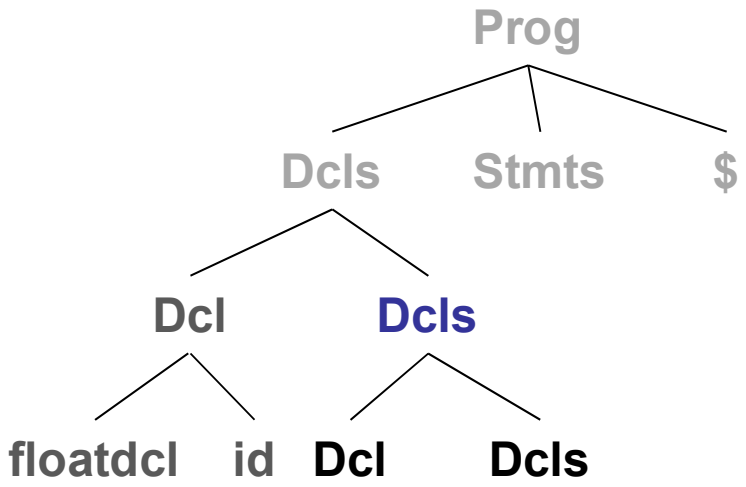
한 번에 한 개의 비단말기호에 대한 생성 규칙 적용

3	⟨Dcl⟩ Dcls Stmts $	2
4	floatdcl id ⟨Dcls⟩ Stmts $	4
5	floatdcl id ⟨Dcl⟩ Dcls Stmts $	2

Which one to choose for the nonterminal <Dcls>?

```
f  b
i  a
```

```
a = 5
b = a + 3.2
p  b
```



Derivation (3/4)

```
 1  Prog  → Dcls Stmts $
 2  Dcls  → Dcl Dcls
 3        | λ
 4  Dcl   → floatdcl id
 5        | intdcl id
 6  Stmts → Stmt Stmts
 7        | λ
 8  Stmt  → id assign Val Expr
 9        | print id
10  Expr  → plus Val Expr
11        | minus Val Expr
12        | λ
13  Val   → id
14        | inum
15        | fnum
```

```
f b
i a
```

```
a = 5
b = a + 3.2
p b
```

Step	Sentential Form	Production Number
1	⟨Prog⟩	
2	**⟨Dcls⟩ Stmts $**	1
3	**⟨Dcl⟩ Dcls** Stmts $	2
4	**floatdcl id** ⟨Dcls⟩ Stmts $	4
5	floatdcl id **⟨Dcl⟩ Dcls** Stmts $	2
6	floatdcl id **intdcl id** ⟨Dcls⟩ Stmts $	5
7	floatdcl id intdcl id ⟨Stmts⟩ $	3
8	floatdcl id intdcl id **⟨Stmt⟩ Stmts** $	6
9	floatdcl id intdcl id **id assign ⟨Val⟩ Expr** Stmts $	8

Derivation (4/4)

1	Prog	→	Dcls Stmts $
2	Dcls	→	Dcl Dcls
3		|	λ
4	Dcl	→	floatdcl id
5		|	intdcl id
6	Stmts	→	Stmt Stmts
7		|	λ
8	Stmt	→	id assign Val Expr
9		|	print id
10	Expr	→	plus Val Expr
11		|	minus Val Expr
12		|	λ
13	Val	→	id
14		|	inum
15		|	fnum

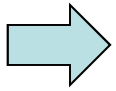
```
f b
i a
```

```
a = 5
b = a + 3.2
p b
```

Step	Sentential Form	Production Number
1	⟨Prog⟩	
2	⟨Dcls⟩ Stmts $	1
3	⟨Dcl⟩ Dcls Stmts $	2
4	floatdcl id ⟨Dcls⟩ Stmts $	4
5	floatdcl id ⟨Dcl⟩ Dcls Stmts $	2
6	floatdcl id intdcl id ⟨Dcls⟩ Stmts $	5
7	floatdcl id intdcl id ⟨Stmts⟩ $	3
8	floatdcl id intdcl id ⟨Stmt⟩ Stmts $	6
9	floatdcl id intdcl id id assign ⟨Val⟩ Expr Stmts $	8
10	floatdcl id intdcl id id assign inum ⟨Expr⟩ Stmts $	14
11	floatdcl id intdcl id id assign inum ⟨Stmts⟩ $	12
12	floatdcl id intdcl id id assign inum ⟨Stmt⟩ Stmts $	6
13	floatdcl id intdcl id id assign inum id assign ⟨Val⟩ Expr Stmts $	8
14	floatdcl id intdcl id id assign inum id assign id ⟨Expr⟩ Stmts $	13
15	floatdcl id intdcl id id assign inum id assign id plus ⟨Val⟩ Expr Stmts $	10
16	floatdcl id intdcl id id assign inum id assign id plus fnum ⟨Expr⟩ Stmts $	15
17	floatdcl id intdcl id id assign inum id assign id plus fnum ⟨Stmts⟩ $	12
18	floatdcl id intdcl id id assign inum id assign id plus fnum ⟨Stmt⟩ Stmts $	6
19	floatdcl id intdcl id id assign inum id assign id plus fnum print id ⟨Stmts⟩ $	9
20	floatdcl id intdcl id id assign inum id assign id plus fnum print id $	7

Do you remember this?

- Keywords
  - **f** , **i** , **p**
- Types :
  - 정수(**i**), 실수(**f**)
- Variables
  - 알파벳 소문자 23자
  - 키워드 3개 제외



Terminal	Input symbol
floatdcl	f
intdcl	i
assign	=
plus	+
minus	-
print	p
id	a, b, c,…
inum	12, 345, …
fnum	0.1, 3.14, …

Token Specification (1/2)

```
1  Prog  → Dcls Stmts $
2  Dcls  → Dcl Dcls
3        | λ
4  Dcl   → floatdcl id
5        | intdcl id
6  Stmts → Stmt Stmts
7        | λ
8  Stmt  → id assign Val Expr
9        | print id
10 Expr  → plus Val Expr
11       | minus Val Expr
12       | λ
13 Val   → id
14       | inum
15       | fnum
```

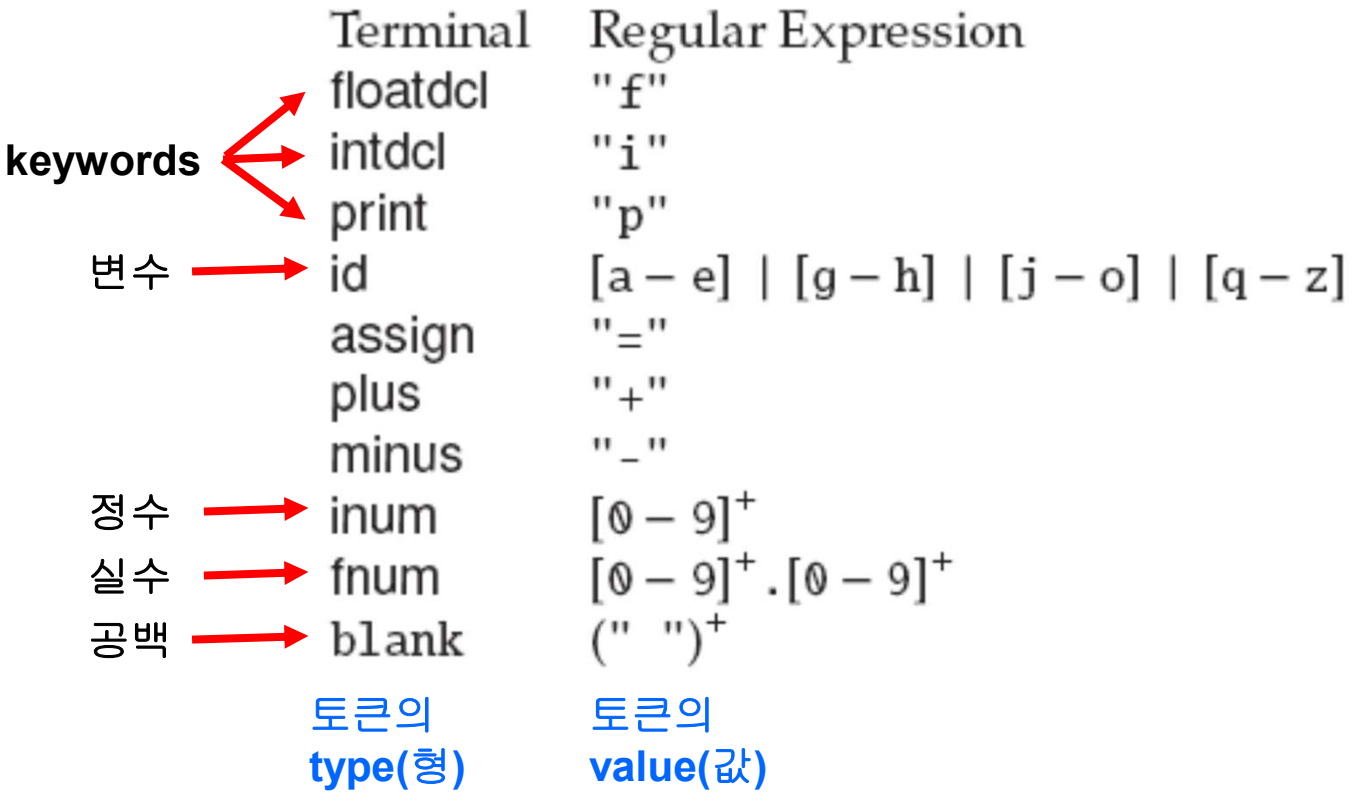
정규 표현

Terminal	Regular Expression
floatdcl	"f"
intdcl	"i"
print	"p"
id	[a – e] | [g – h] | [j – o] | [q – z]
assign	"="
plus	"+"
minus	"-"
inum	$[0-9]^+$
fnum	$[0-9]^+.[0-9]^+$
blank	$(" ")^+$

정규 표현에 **meta symbol**을 사용할 수 있다.

Formal definition of ac tokens.

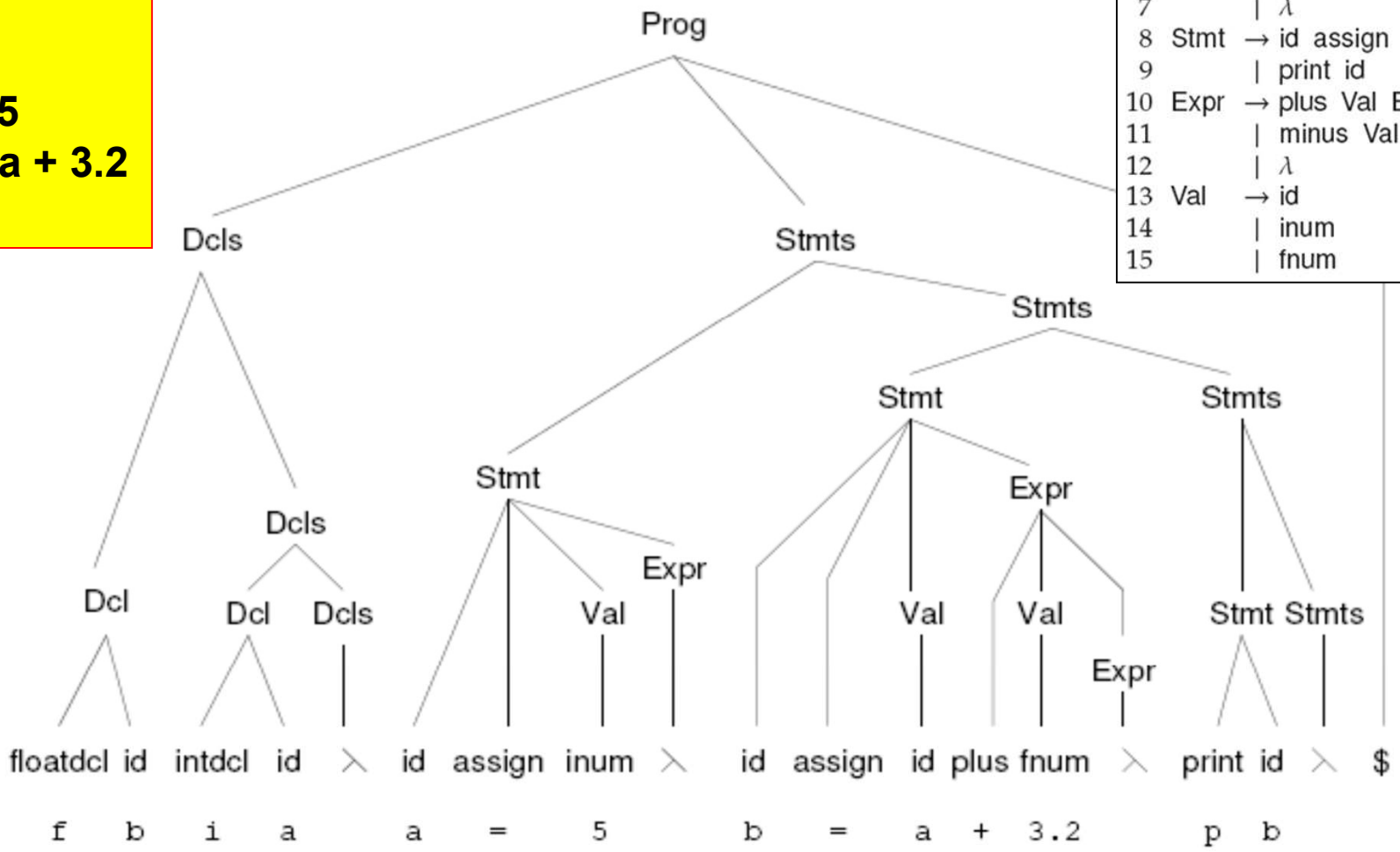
Token Specification (2/2)



Parse tree

```
f b
i a
a = 5
b = a + 3.2
p b
```

```
1  Prog  → Dcls Stmts $
2  Dcls  → Dcl Dcls
3        | λ
4  Dcl   → floatdcl id
5        | intdcl id
6  Stmts → Stmt Stmts
7        | λ
8  Stmt  → id assign Val Expr
9        | print id
10 Expr  → plus Val Expr
11       | minus Val Expr
12       | λ
13 Val   → id
14       | inum
15       | fnum
```



An ac program and its parse tree.