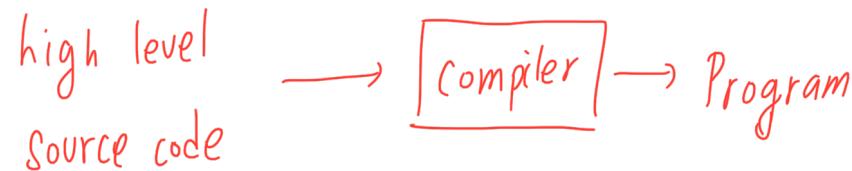


2. Compiler Structure



학습 목표 및 목차

■ 목표

- 컴파일러의 논리적 구조를 이해할 수 있다.
- 간단한 컴파일러의 예를 통하여 컴파일러의 전체 구조를 이해할 수 있다.
- 컴파일러의 물리적 구조를 이해할 수 있다.

■ 목차

- 컴파일러의 논리적 구조
- 컴파일러의 물리적 구조

2.1 컴파일러의 논리적 구조

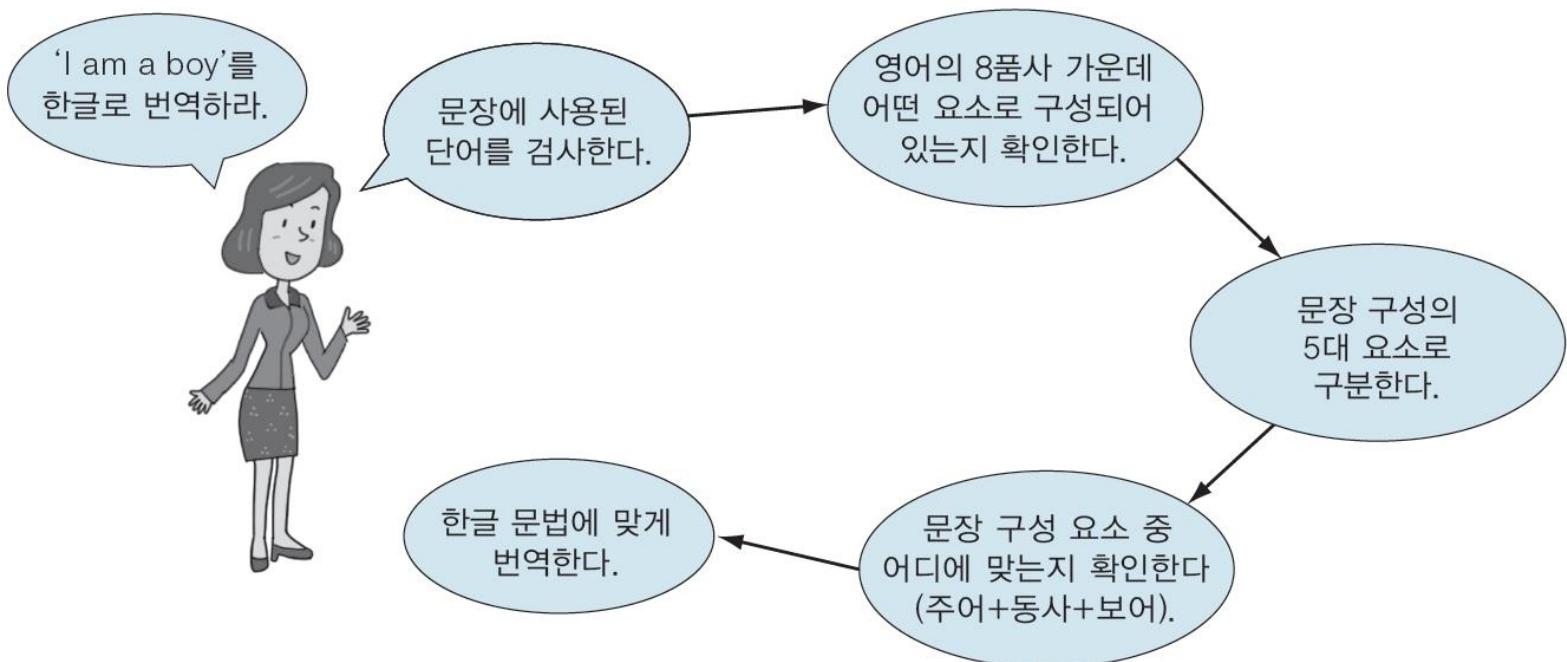


그림 2-1 영어를 한글로 번역

2.1 컴파일러의 논리적 구조

- 1. 어휘 분석 (Lexical Analysis)
 - 문장이 어떤 요소로 구성되어 있는지 파악하기 위해 문장에 사용된 단어를 검사
 - 예를 들어, 문장에 I, am, a, boy라는 네 가지 단어가 사용된 것을 알아냄
- 2. 구문 분석 (Syntax Analysis)
 - 문장의 형식을 분석
 - 단어들이 어떠한 8품사(명사, 대명사, 동사, 형용사, 부사, 전치사, 접속사, 감탄사) 중에 어디에 속하는지 확인하고, 문장의 5대 요소인 주어, 동사, 목적어, 보어, 수식어 등으로 구분
 - 예를 들어, 'I am a boy'는 주어+동사+보어로 구성되어 있다는 것을 알아냄
- 3. 의미 분석 (Semantic Analysis)
 - 단어대 단어의 번역이 아닌 한글 문법에 맞게 번역
- 4. 코드 최적화 및 코드 생성 (Code optimization and generation)
 - 한글 단어 가운데 더 알맞은 단어로 번역하거나 문장들의 순서를 조절하고, 마지막으로 한글로 번역

2.1 컴파일러의 논리적 구조

- 컴파일러도 영어 문장을 한글로 번역하는 것과 비슷한 단계 거쳐서 번역
- 1. 어휘 분석
 - C 언어에서 사용하는 의미 있는 단어를 '토큰 token'이라고 한다.
 - C 언어에서 어떤 토큰이 사용되었는지 구분
- 2. 구문 분석
 - 문법을 보면서 토큰들이 문법에 맞는지 검사 *세미콜론 등.*
- 3. 의미 분석
 - 형식 언어는 의미를 가지지 않음. 따라서 형 (type) 검사 등을 실시
- 4. 코드 최적화 및 코드 생성
 - 목적 기계(target machine)의 특성을 고려한 목적 코드 생성

X86.

ARM

⋮

2.1 컴파일러의 논리적 구조

▪ 컴파일러의 구조에 대한 견해 :

2-path 방식 \Rightarrow 전단부(front-end), 후단부(Back-end) 구조

- 전단부(Front-end)-후단부(Back-end) : 기계 독립(machine independent) or 기계 종속(dependent), 구현측면

▪ 컴파일러의 구체적 구조

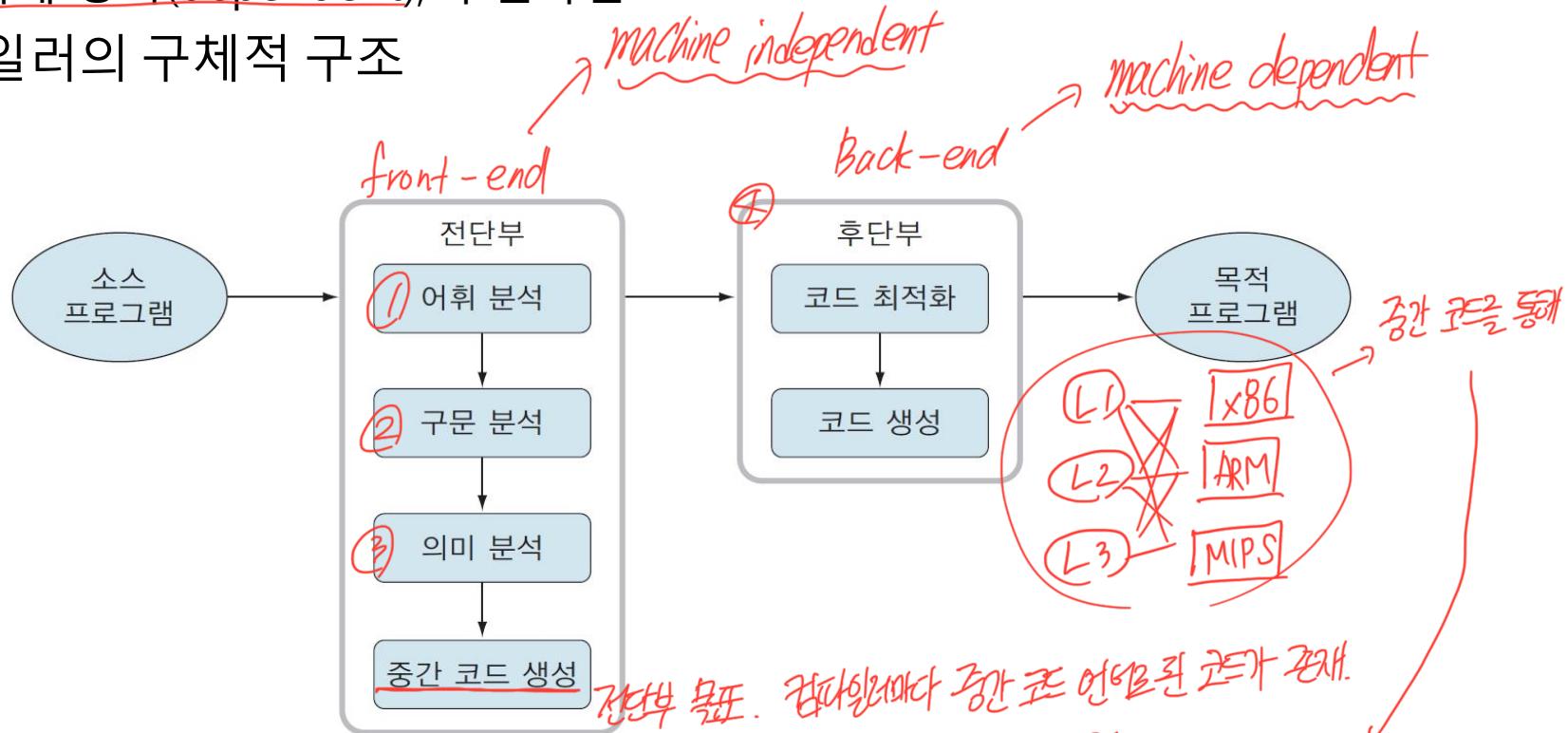


그림 2-2 컴파일러를 전단부와 후단부로 나눈 논리적 구조

전단부/후단부를 분리하여
언어 - 컴파일러 짜당
컴파일러를 할당이 만들지 않아도
된다.

2.1 컴파일러의 논리적 구조

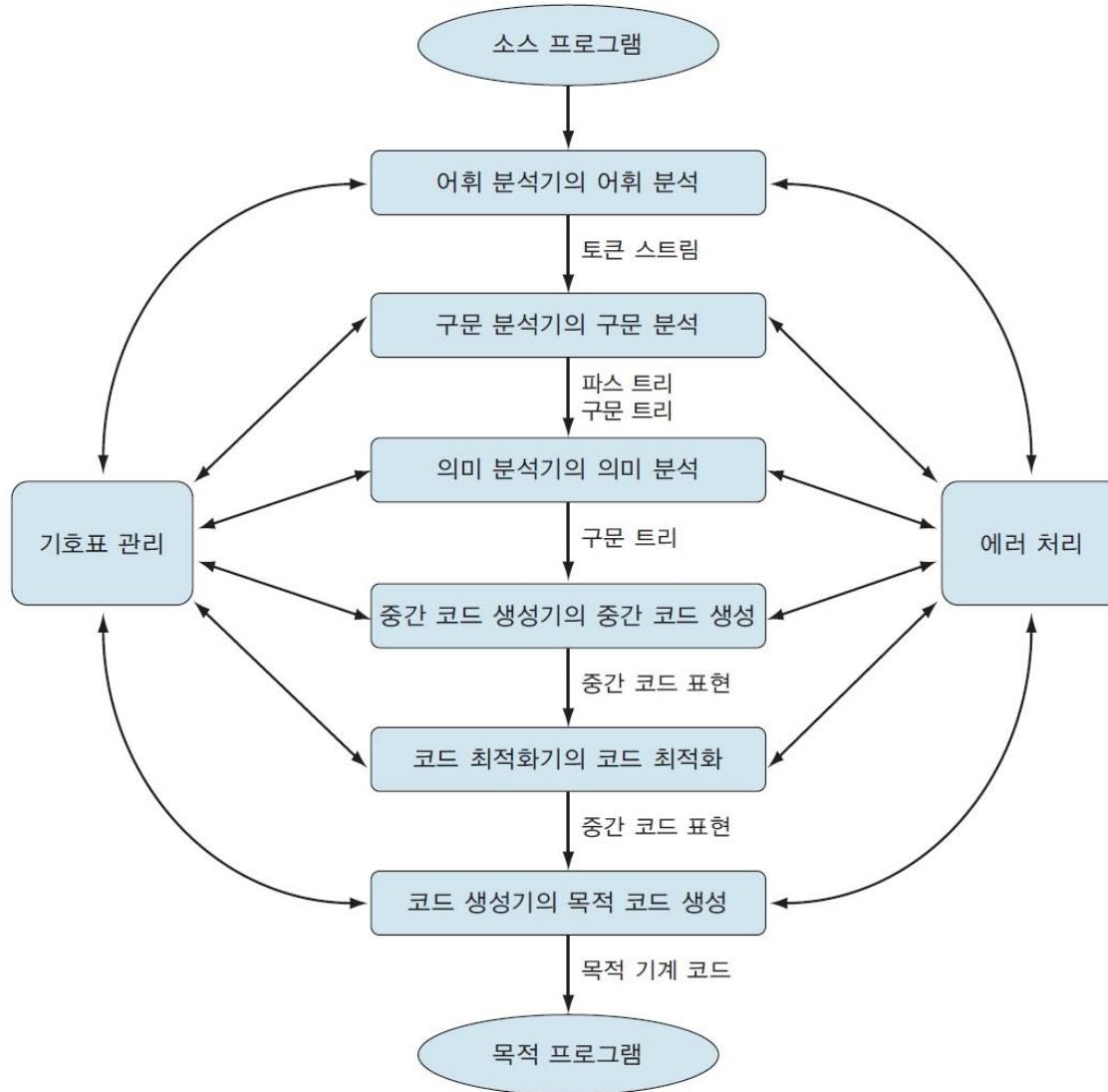


그림 2-3 컴파일러의 논리적 구조

2.1 컴파일러의 논리적 구조

■ 아주 간단한 C 문법 (EBNF 형태)

- ① $\langle \text{Sub C} \rangle ::= \langle \text{assign-st} \rangle$
- ② $\langle \text{assign-st} \rangle ::= \langle \text{lhs} \rangle = \langle \text{exp} \rangle;$
- ③ $\langle \text{lhs} \rangle ::= \langle \text{variable} \rangle$
- ④ $\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{term} \rangle \mid \langle \text{exp} \rangle - \langle \text{term} \rangle \mid \langle \text{term} \rangle$
- ⑤ $\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{term} \rangle / \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$
- ⑥ $\langle \text{factor} \rangle ::= \langle \text{variable} \rangle \mid \langle \text{number} \rangle \mid (\langle \text{exp} \rangle)$
- ⑦ $\langle \text{variable} \rangle ::= \langle \text{ident} \rangle$
- ⑧ $\langle \text{ident} \rangle ::= (\langle \text{letter} \rangle \mid _) \{ \langle \text{letter} \rangle \mid \langle \text{digit} \rangle \mid _ \}$
- ⑨ $\langle \text{number} \rangle ::= \{ \langle \text{digit} \rangle \}$
- ⑩ $\langle \text{letter} \rangle ::= a \mid \cdots \mid z$
- ⑪ $\langle \text{digit} \rangle ::= 0 \mid 1 \mid \cdots \mid 9$

그림 2-4 아주 간단한 C 문법

2.1 컴파일러의 논리적 구조

- [예제 2-1] 치환문 $ni = (po - 60)$; 이 주어진 문법에 맞는 문장인지 확인해보자.
- 이를 확인하는 방법은 크게 두 가지가 있는데 여기서는 좌단 유도(leftmost derivation)하는 방법으로 확인해본다. 유도는 생성 규칙의 왼쪽을 오른쪽으로 대체하는 것이며, 좌단 유도는 가장 왼쪽 기호부터 유도하는 것을 말한다
- [풀이]

$\langle \text{Sub C} \rangle \Rightarrow \langle \text{assign-st} \rangle$ (\because 생성 규칙 ① 적용)

$\Rightarrow \langle \text{lhs} \rangle = \langle \text{exp} \rangle$; (\because 생성 규칙 ② 적용)

$\Rightarrow \langle \text{variable} \rangle = \langle \text{exp} \rangle$; (\because 생성 규칙 ③ 적용)

$\Rightarrow \langle \text{ident} \rangle = \langle \text{exp} \rangle$; (\because 생성 규칙 ⑦ 적용)

$\Rightarrow (\langle \text{letter} \rangle | _) \{ \langle \text{letter} \rangle | \langle \text{digit} \rangle | _ \} = \langle \text{exp} \rangle$; (\because 생성 규칙 ⑧ 적용)

$\Rightarrow n \{ \langle \text{letter} \rangle | \langle \text{digit} \rangle | _ \} = \langle \text{exp} \rangle$; (\because 생성 규칙 ⑩ 적용)

$\Rightarrow ni = \langle \text{exp} \rangle$; (\because 생성 규칙 ⑩ 적용)

$\Rightarrow ni = \langle \text{term} \rangle$; (\because 생성 규칙 ④ 적용)

exp = expression

- ① $\langle \text{Sub C} \rangle ::= \langle \text{assign-st} \rangle$
- ② $\langle \text{assign-st} \rangle ::= \langle \text{lhs} \rangle = \langle \text{exp} \rangle$;
- ③ $\langle \text{lhs} \rangle ::= \langle \text{variable} \rangle$
- ④ $\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{term} \rangle | \langle \text{exp} \rangle - \langle \text{term} \rangle | \langle \text{term} \rangle$
- ⑤ $\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle | \langle \text{term} \rangle / \langle \text{factor} \rangle | \langle \text{factor} \rangle$
- ⑥ $\langle \text{factor} \rangle ::= \langle \text{variable} \rangle | \langle \text{number} \rangle | (\langle \text{exp} \rangle)$
- ⑦ $\langle \text{variable} \rangle ::= \langle \text{ident} \rangle$
- ⑧ $\langle \text{ident} \rangle ::= (\langle \text{letter} \rangle | _) \{ \langle \text{letter} \rangle | \langle \text{digit} \rangle | _ \}$
- ⑨ $\langle \text{number} \rangle ::= \{ \langle \text{digit} \rangle \}$
- ⑩ $\langle \text{letter} \rangle ::= a | \dots | z$
- ⑪ $\langle \text{digit} \rangle ::= 0 | 1 | \dots | 9$

그림 2-4 아주 간단한 C 문법

2.1 컴파일러의 논리적 구조

⇒ ni = <factor>; (∴ 생성 규칙 ⑤ 적용)

⇒ ni = (<exp>); (∴ 생성 규칙 ⑥ 적용)

⇒ ni = (<exp>-<term>); (∴ 생성 규칙 ④ 적용)

⇒ ni = (<term>-<term>); (∴ 생성 규칙 ④ 적용)

⇒ ni = (<factor>-<term>); (∴ 생성 규칙 ⑤ 적용)

⇒ ni = (<variable>-<term>); (∴ 생성 규칙 ⑥ 적용)

⇒ ni = (<ident>-<term>); (∴ 생성 규칙 ⑦ 적용)

⇒ ni = ((<letter>|_)<letter>|<digit>|_)-<term>); (∴ 생성 규칙 ⑧ 적용)

⇒ ni = (p<letter>|<digit>|_)-<term>); (∴ 생성 규칙 ⑩ 적용)

⇒ ni = (po - <term>); (∴ 생성 규칙 ⑩ 적용)

⇒ ni = (po - <factor>); (∴ 생성 규칙 ⑤ 적용)

⇒ ni = (po - <number>); (∴ 생성 규칙 ⑥ 적용)

⇒ ni = (po - {<digit>}); (∴ 생성 규칙 ⑨ 적용)

⇒ ni = (po - 60); (∴ 생성 규칙 11 적용)

치환문 ni = (po - 60);이 주어진 문법에 맞는 문장이다.

- ① <Sub C> ::= <assign-st>
- ② <assign-st> ::= <lhs> = <exp>;
- ③ <lhs> ::= <variable>
- ④ <exp> ::= <exp>+<term> | <exp>-<term> | <term>
- ⑤ <term> ::= <term>*<factor> | <term>/<factor> | <factor>
- ⑥ <factor> ::= <variable> | <number> | (<exp>)
- ⑦ <variable> ::= <ident>
- ⑧ <ident> ::= (<letter>|_)<letter>|<digit>|_
- ⑨ <number> ::= {<digit>}
- ⑩ <letter> ::= a | ⋯ | z
- ⑪ <digit> ::= 0 | 1 | ⋯ | 9

그림 2-4 아주 간단한 C 문법

2.1 컴파일러의 논리적 구조

- 치환문 $ni = ba * po - 60 + ni / (abc + 50);$ 에 대한 컴파일러 도식화

적은 품질

$$ni = ba * po - 60 + ni / (abc + 50);$$

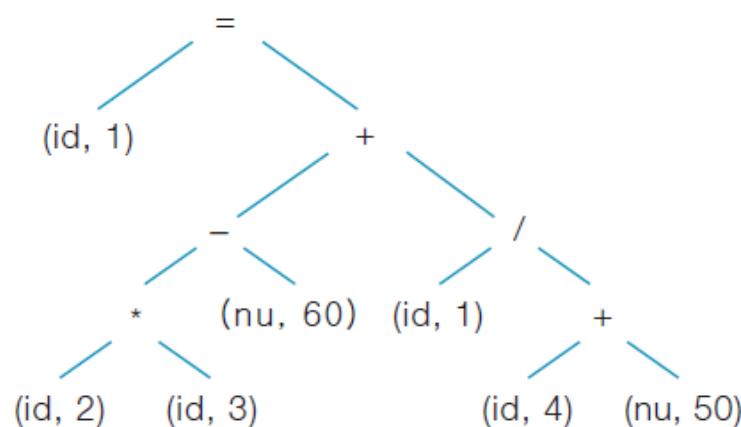
↓
어휘 분석기의 어휘 분석

(id, 1) (=, ~) (id, 2) (*, ~) (id, 3) (-, ~) (nu, 60) (+, ~)
(id, 1) (/ , ~) ((, ~) (id, 4) (+, ~) (nu, 50) (, ~) (;, ~)

기호표

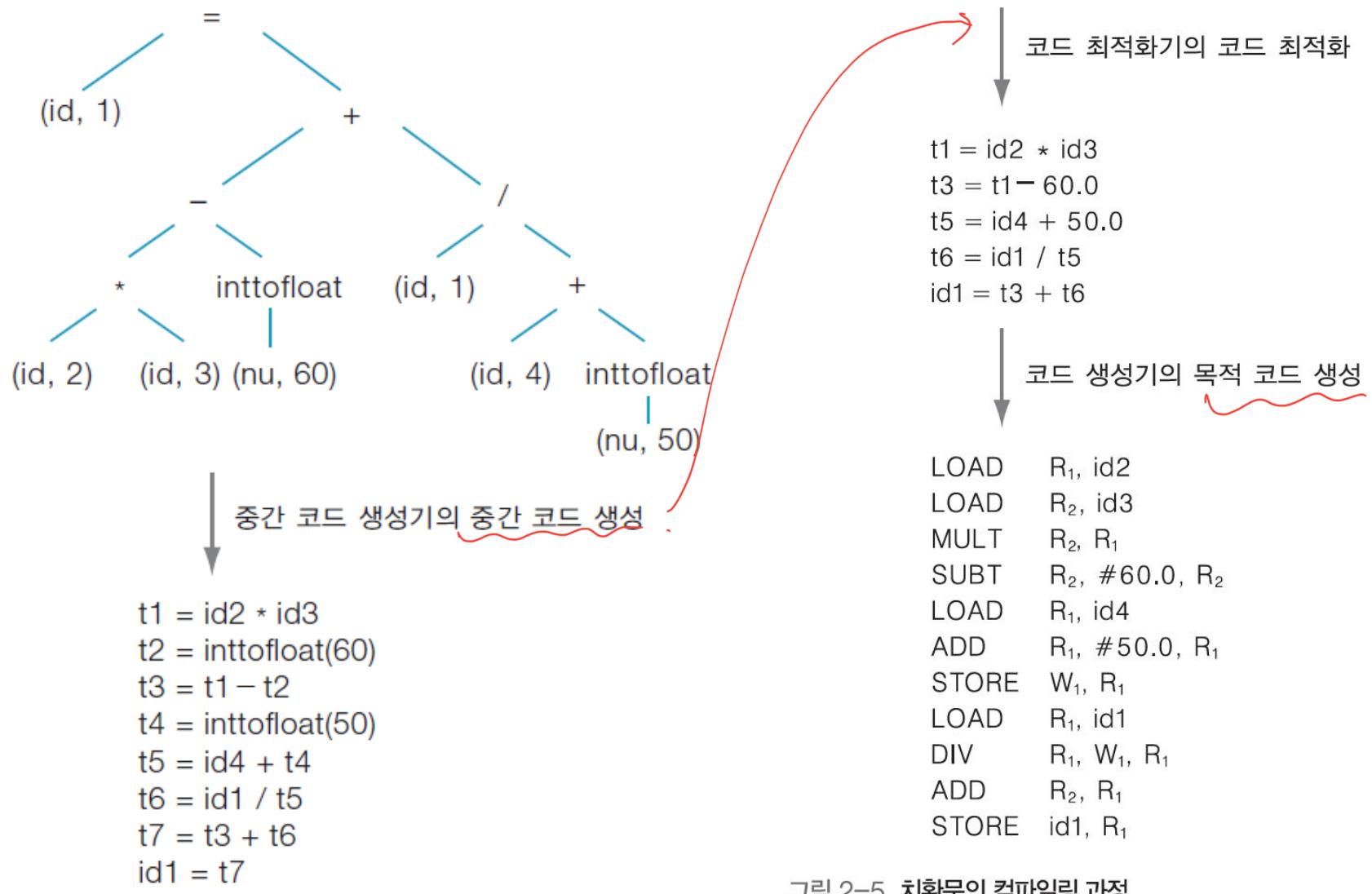
1	ni	...
2	ba	...
3	po	...
4	abc	...

파스 트리
구문 트리



↓
의미 분석기의 의미 분석

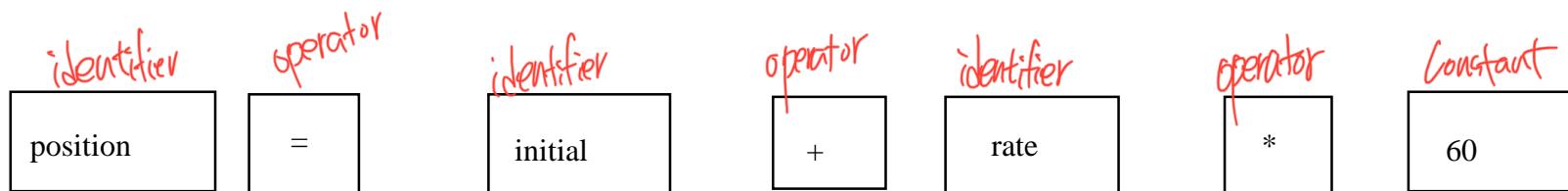
2.1 컴파일러의 논리적 구조



2.1 컴파일러의 논리적 구조

■ 어휘분석(lexical analysis, scanning)

- 원시 프로그램을 읽어 들여 토큰(token)이라는 의미 있는 문법적 단위(syntactic entity)로 분리
- 어휘분석을 담당하는 도구(tool)를 어휘 분석기(lexical analyzer) 혹은 스캐너(scanner)라고 부른다.
- 토큰 : 문법적으로 의미있는 최소 단위로 문법에서 터미널 기호임

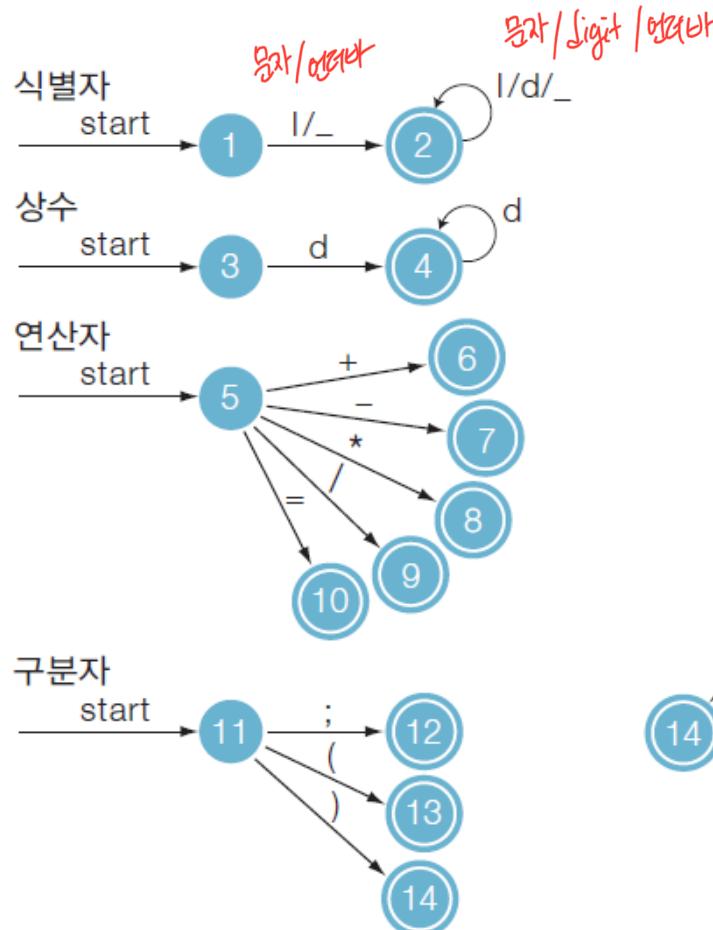


■ 토큰의 종류

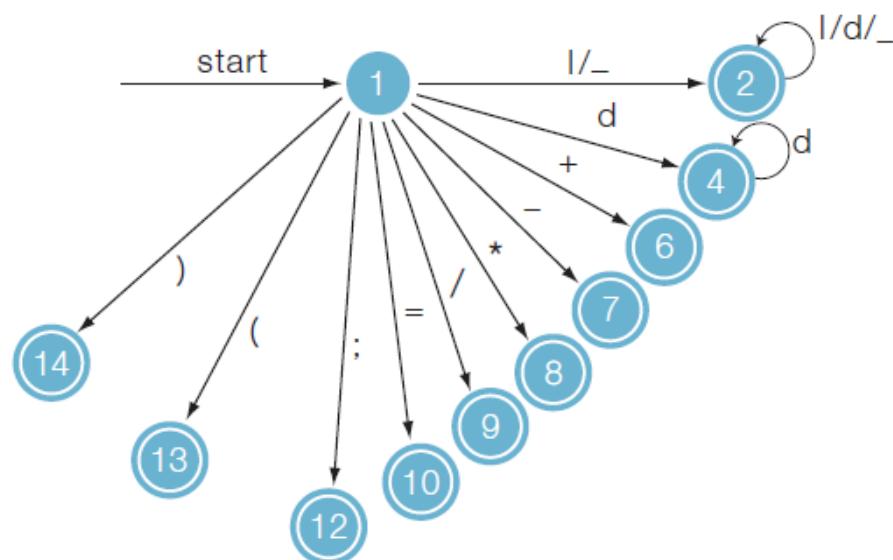
- 식별자(identifier) (position, initial, rate,)
- 예약어(key word) (IF, WHILE, ...) : reserved word
- 상수(constant) (60, “ KIM “) : numerical, literal
- 연산자(operator) (+, *, %,)
- 구분자(delimiter) ([], ;)

2.1 컴파일러의 논리적 구조

- 이들을 받아들이는 유한 오토마타는 다음과 같다.



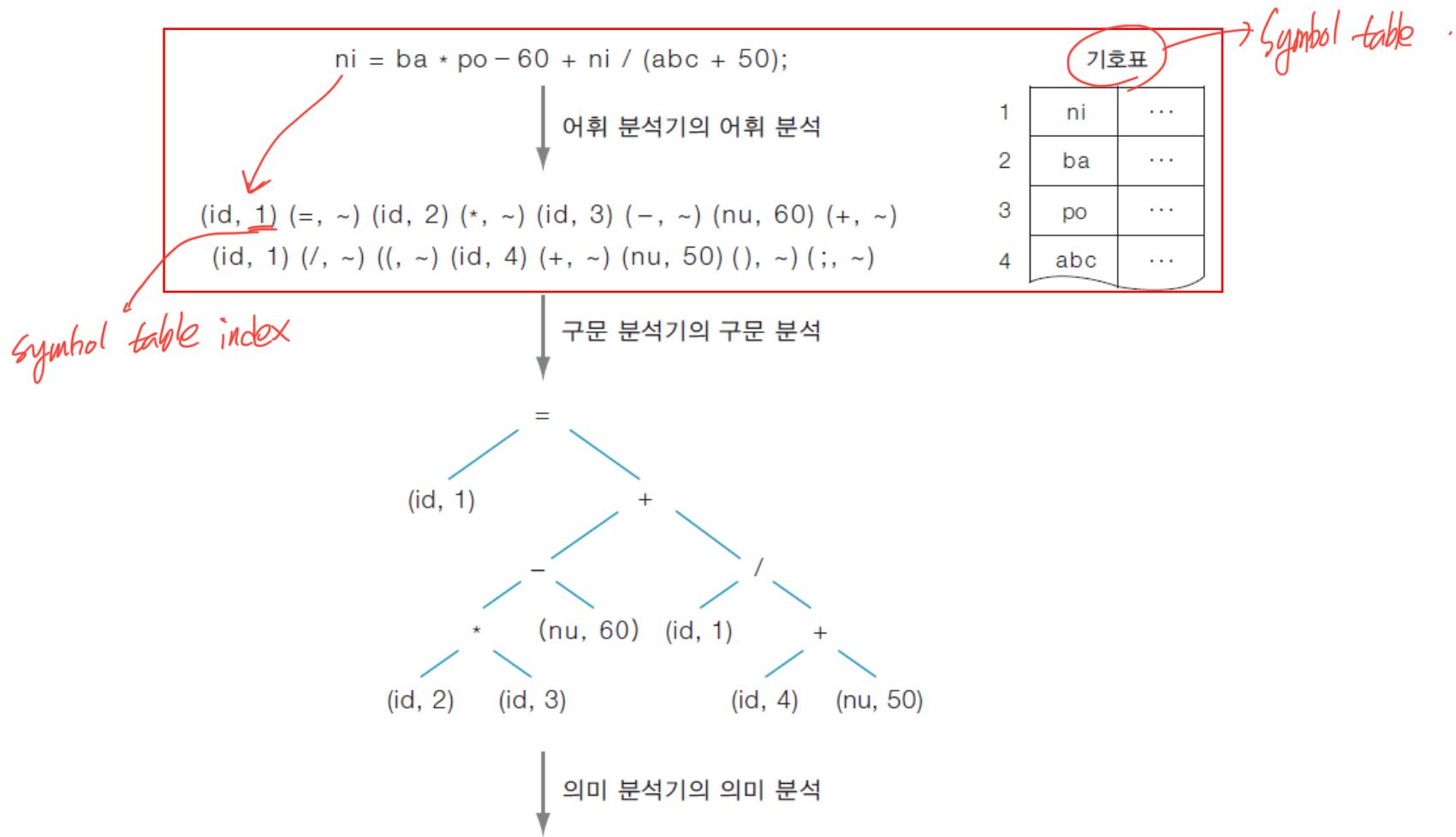
(a) 각각의 토큰에 대한 어휘 분석기



(b) 전체 토큰에 대한 어휘 분석기

2.1 컴파일러의 논리적 구조

- 토큰에 대한 표현은 (토큰 번호, 토큰 값)의 순서쌍으로 표현
- 주석(comment)도 이 과정에서 처리



2.1 컴파일러의 논리적 구조

▪ 구문분석(syntax-analysis, parsing)

- 어휘분석 단계로부터 토큰들을 받아 이 토큰들이 주어진 문법(grammar)에 맞는지를 검사하고 올바른 문장에 대해서는 그 문장에 대한 파스 트리(parse tree)를 출력하고 올바르지 못한 문장에 대해서는 에러 메시지(error message)를 출력하는 과정을 구문 분석(syntax analysis) 혹은 파싱(parsing).
- 구문분석을 담당하는 도구(tool)를 구문 분석기(syntax analyzer) 혹은 파서(parser)라고 부른다.
- 파스 트리(parse tree)는 토큰을 터미널 노드(terminal node)로 하는 트리(tree)
- 파싱표에 사용하기 위해 생성 규칙 번호를 다음과 같이 정의.

- | | | |
|---|---|---|
| 0. $\langle S' \rangle ::= \langle Sub C \rangle$ | 1. $\langle Sub C \rangle ::= \langle as \rangle$ | 2. $\langle as \rangle ::= \langle lh \rangle = \langle ex \rangle$ |
| 3. $\langle lh \rangle ::= \langle va \rangle$ | 4. $\langle ex \rangle ::= \langle ex \rangle + \langle te \rangle$ | 5. $\langle ex \rangle ::= \langle ex \rangle - \langle te \rangle$ |
| 6. $\langle ex \rangle ::= \langle te \rangle$ | 7. $\langle te \rangle ::= \langle te \rangle * \langle fa \rangle$ | 8. $\langle te \rangle ::= \langle te \rangle / \langle fa \rangle$ |
| 9. $\langle te \rangle ::= \langle fa \rangle$ | 10. $\langle fa \rangle ::= \langle va \rangle$ | 11. $\langle fa \rangle ::= \langle nu \rangle$ |
| 12. $\langle fa \rangle ::= (\langle ex \rangle)$ | 13. $\langle va \rangle ::= \langle id \rangle$ | |

2.1 컴파일러의 논리적 구조

- $ni = ba * po - 60 + ni / (abc + 50);$ 에 대한 파스 트리

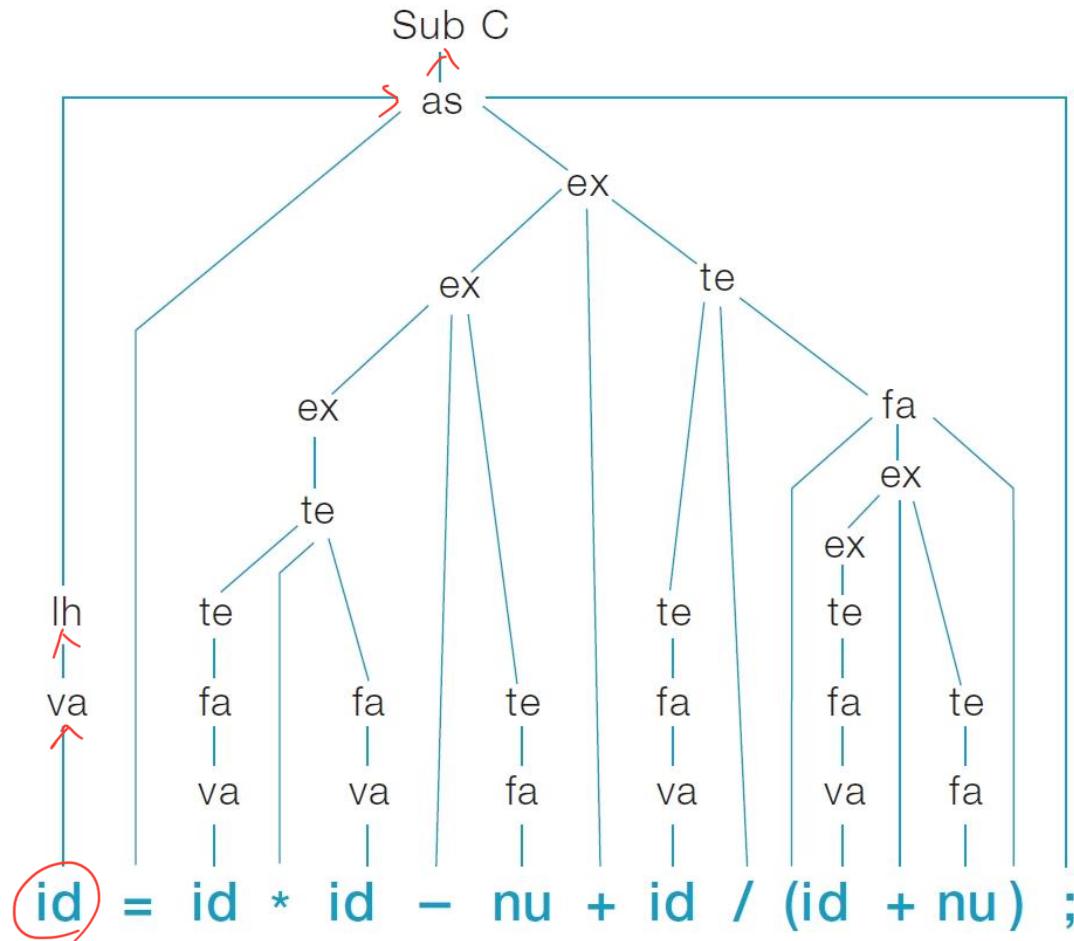


그림 2-7 치환문 $ni = ba * po - 60 + ni / (abc + 50);$ 의 파스 트리

2.1 컴파일러의 논리적 구조

Syntax analysis 이란 자세히 설명.

- 파스 트리에 대한 구문 트리

파스 트리 → 구문 트리

정한 rule을 통해 처리 불가 → Syntax error

$$ni = ba * po - 60 + ni / (abc + 50);$$

어휘 분석기의 어휘 분석

$$\begin{aligned} & (id, 1) (=, \sim) (id, 2) (*, \sim) (id, 3) (-, \sim) (nu, 60) (+, \sim) \\ & (id, 1) (/ , \sim) ((, \sim) (id, 4) (+, \sim) (nu, 50) (), \sim) (;, \sim) \end{aligned}$$

기호표	
1	ni
2	ba
3	po
4	abc

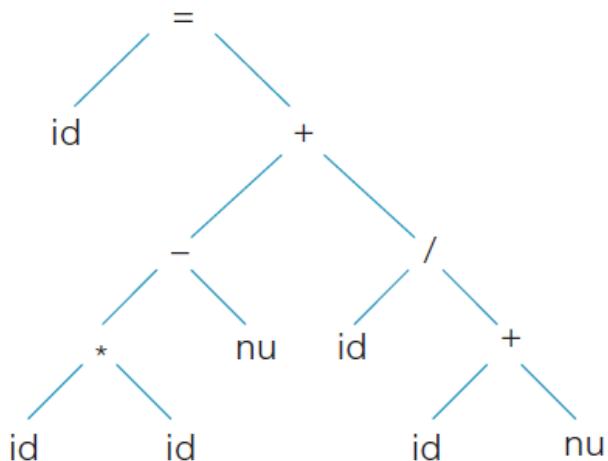
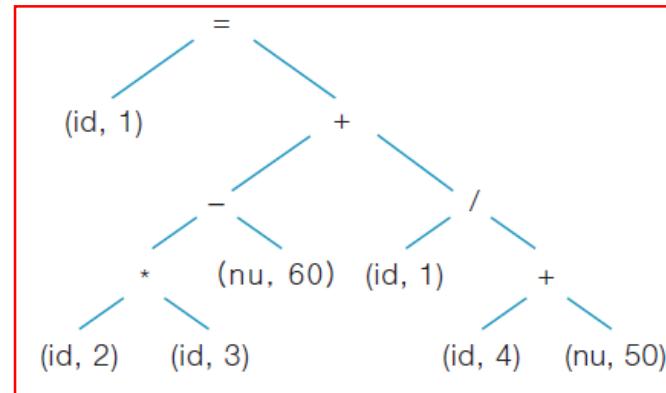


그림 2-8 [그림 2-7]의 파스 트리에 대한 구문 트리



의미 분석기의 의미 분석

2.1 컴파일러의 논리적 구조

- 실제 parser는 문법에 대한 다음과 같은 파싱표를 작성하고, 이를 토대로 구문 분석을 실시한다.

표 2-1 [그림 2-4]의 문법에 대한 SLR 파싱표 → parser 자동으로 자세히 설명.

상태	구문 분석기의 행동												GOTO 함수					
	id	nu	+	-	*	/	;	=	()	\$	su	as	lh	ex	te	fa	va
0	s10	s8							s9			1	2	3	5	6	7	4
1											acc							
2											r1							
3								s11										
4			r10	r10	r10	r10	r10	r3		r10								
5			s12	s13														
6			r6	r6	s14	s15	r6				r6							
7			r9	r9	r9	r9	r9				r9							
8			r11	r11	r11	r11	r11				r11							
9	s10	s8							s9						16	6	7	17

2.1 컴파일러의 논리적 구조

10			r13	r13	r13	r13	r13	r13																
11	s10	s8							s9									18	6	7	17			
12	s10	s8							s9									19	7	17				
13	s10	s8							s9									20	7	17				
14	s10	s8							s9									21	17					
15	s10	s8							s9									22	17					
16			s12	s13						s23														
17			r10	r10	r10	r10	r10			r10														
18			s12	s13				s24																
19			r4	r4	s14	s15	r4			r4														
20			r5	r5	s14	s15	r5			r5														
21			r7	r7	r7	r7	r7			r7														
22			r8	r8	r8	r8	r8			r8														
23			r12	r12	r12	r12	r12			r12														
24										r2														

2.1 컴파일러의 논리적 구조

표 2-2 치환문 $ni = (((ba * po) - 60) + (ni / (abc + 50)))$ 의 파싱 과정

단계	스택	입력 기호	구문 분석 내용
0	0	$id=id*id-nu+id/(id+nu)\$$	shift 10
1	0id10	$=id=id-nu+id/(id+nu)\$$	reduce 13
2	0va	$=id=id-nu+id/(id+nu)\$$	goto 4
3	0va4	$=id=id-nu+id/(id+nu)\$$	reduce 3
4	0lh	$=id=id-nu+id/(id+nu)\$$	goto 3
5	0lh3	$=id=id-nu+id/(id+nu)\$$	shift 11
6	0lh3=11	$id=id-nu+id/(id+nu)\$$	shift 10
7	0lh3=11id10	$*id-nu+id/(id+nu)\$$	reduce 13
8	0lh3=11va	$*id-nu+id/(id+nu)\$$	goto 17
9	0lh3=11va17	$*id-nu+id/(id+nu)\$$	reduce 10
10	0lh3=11fa	$*id-nu+id/(id+nu)\$$	goto 7
11	0lh3=11fa7	$*id-nu+id/(id+nu)\$$	reduce 9
12	0lh3=11te	$*id-nu+id/(id+nu)\$$	goto 6
13	0lh3=11te6	$*id-nu+id/(id+nu)\$$	shift 14
14	0lh3=11te6*14	$id-nu+id/(id+nu)\$$	shift 10
15	0lh3=11te6*14id10	$-nu+id/(id+nu)\$$	reduce 13
16	0lh3=11te6*14va	$-nu+id/(id+nu)\$$	goto 17
17	0lh3=11te6*14va17	$-nu+id/(id+nu)\$$	reduce 10
18	0lh3=11te6*14fa	$-nu+id/(id+nu)\$$	goto 21
19	0lh3=11te6*14fa21	$-nu+id/(id+nu)\$$	reduce 7
20	0lh3=11te	$-nu+id/(id+nu)\$$	goto 6

2.1 컴파일러의 논리적 구조

단계	스택	입력 기호	구문 분석 내용
21	0lh3=11te6	-nu+id/(id+nu)\$	reduce 6
22	0lh3=11ex	-nu+id/(id+nu)\$	goto 18
23	0lh3=11ex18	-nu+id/(id+nu)\$	shift 13
24	0lh3=11ex18-13	nu+id/(id+nu)\$	shift 8
25	0lh3=11ex18-13nu8	+id/(id+nu)\$	reduce 11
26	0lh3=11ex18-13fa	+id/(id+nu)\$	goto 7
27	0lh3=11ex18-13fa7	+id/(id+nu)\$	reduce 9
28	0lh3=11ex18-13te	+id/(id+nu)\$	goto 20
29	0lh3=11ex18-13te20	+id/(id+nu)\$	reduce 5
30	0lh3=11ex	+id/(id+nu)\$	goto 18
31	0lh3=11ex18	+id/(id+nu)\$	shift 12
32	0lh3=11ex18+12	id/(id+nu)\$	shift 10
33	0lh3=11ex18+12id10	/(id+nu)\$	reduce 13
34	0lh3=11ex18+12va	/(id+nu)\$	goto 17
35	0lh3=11ex18+12va17	/(id+nu)\$	reduce 10
36	0lh3=11ex18+12fa	/(id+nu)\$	goto 7
37	0lh3=11ex18+12fa7	/(id+nu)\$	reduce 9
38	0lh3=11ex18+12te	/(id+nu)\$	goto 19
39	0lh3=11ex18+12te19	/(id+nu)\$	shift 15

2.1 컴파일러의 논리적 구조

단계	스택	입력 기호	구문 분석 내용
40	0lh3=11ex18+12te19/15	(id+nu)\$	shift 9
41	0lh3=11ex18+12te19/15(9	id+nu)\$	shift 10
42	0lh3=11ex18+12te19/15(9id10	+nu)\$	reduce 13
43	0lh3=11ex18+12te19/15(9va	+nu)\$	goto 17
44	0lh3=11ex18+12te19/15(9va17	+nu)\$	reduce 10
45	0lh3=11ex18+12te19/15(9fa	+nu)\$	goto 7
46	0lh3=11ex18+12te19/15(9fa7	+nu)\$	reduce 9
47	0lh3=11ex18+12te19/15(9te	+nu)\$	goto 6
48	0lh3=11ex18+12te19/15(9te6	+nu)\$	reduce 6
49	0lh3=11ex18+12te19/15(9ex	+nu)\$	goto 16
50	0lh3=11ex18+12te19/15(9ex16	+nu)\$	shift 12
51	0lh3=11ex18+12te19/15(9ex16+12	nu)\$	shift 8
52	0lh3=11ex18+12te19/15(9ex16+12nu8)\$	reduce 11
53	0lh3=11ex18+12te19/15(9ex16+12fa)\$	goto 7
54	0lh3=11ex18+12te19/15(9ex16+12fa7)\$	reduce 9
55	0lh3=11ex18+12te19/15(9ex16+12te)\$	goto 19

2.1 컴파일러의 논리적 구조

단계	스택	입력 기호	구문 분석 내용
56	0lh3=11ex18+12te19/15(9ex16+12te19)\$	reduce 4
57	0lh3=11ex18+12te19/15(9ex)\$	goto 16
58	0lh3=11ex18+12te19/15(9ex16)\$	shift 23
59	0lh3=11ex18+12te19/15(9ex16)23	\$	reduce 12
60	0lh3=11ex18+12te19/15fa	\$	goto 22
61	0lh3=11ex18+12te19/15fa22	\$	reduce 8
62	0lh3=11ex18+12te	\$	goto 19
63	0lh3=11ex18+12te19	\$	reduce 4
64	0lh3=11ex	\$	goto 18
65	0lh3=11ex18	\$	shift 24
66	0lh3=11ex18;24	\$	reduce 2
67	0as	\$	goto 2
68	0as2	\$	reduce 1
69	0su	\$	goto 1
70	0su1	\$	accept

60

- $ni = ba * po - 60 + ni / (abc + 50);$ 은 주어진 문법에 맞는 문장임을 알 수 있다.

2.1 컴파일러의 논리적 구조

■ 의미분석(semantic-analysis)

- 원시 프로그램의 의미적 오류를 검사하고 계속되는 코드생성 단계를 위한 정보를 모으는 일
- 의미 분석을 담당하는 도구(tool)를 의미 분석기(semantic analyzer)
- 의미 분석 단계에서 가장 중요한 기능 중의 하나는 형 검사(type checking)
 - 정수와 실수의 일반적 연산을 허용하는데 이때의 의미 분석기는 연산을 수행하기 전에 정수를 실수로 바꾸어 주는 작업.
 - 예를 들어 a+b와 같은 산술식을 생각해보자.
 - a와 b의 자료형을 각각 int형과 float형이라면 형 고정 연산에서는 에러.
 - 반면에 일반적 연산에서는 a나 b의 형을 변환하여 연산을 허용. 형 변환(type conversion)이란 주어진 자료형의 값을 다른 자료형의 값으로 변환하는 것을 의미, 이에는 시스템에서 자동으로 형을 변환하는 묵시적 형 변환(implicit type conversion)과 프로그래머가 명시적으로 형 변환을 요구하는 명시적 형 변환(explicit type conversion).
 - 명시적 변환은 프로그래머가 명령문으로 요구한 형 변환으로 캐스트(cast) 연산. 묵시적 형 변환은 시스템에서 자동으로 형을 변환하는 것으로 자동 변환(automatic type conversion) 또는 강제 변환(coercion)

2.1 컴파일러의 논리적 구조

의미 분석 단계에서 자세히 다룬 예정.

- [예 2-3] 어떤 언어가 일반적 연산을 허용하고 정수형인 int와 실수형인 float가 있는 경우, 정수형을 실수형으로 변환하여 계산한다고 가정하고, 치환문 $ni = ba * po - 60 + ni / (abc + 50)$;에서 나타난 식별자 ni, ba, po, ni, abc는 모두 실수형이라고 가정한다. 이때 [그림 2-8]의 구문 트리를 가지고 의미 분석하면 다음과 같다.
 - 여기서 inttofloat는 정수형을 실수형으로 변환하는 함수.

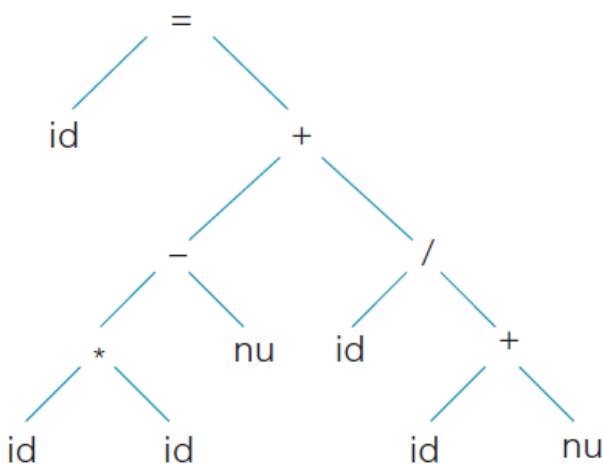


그림 2-8 [그림 2-7]의 파스 트리에 대한 구문 트리

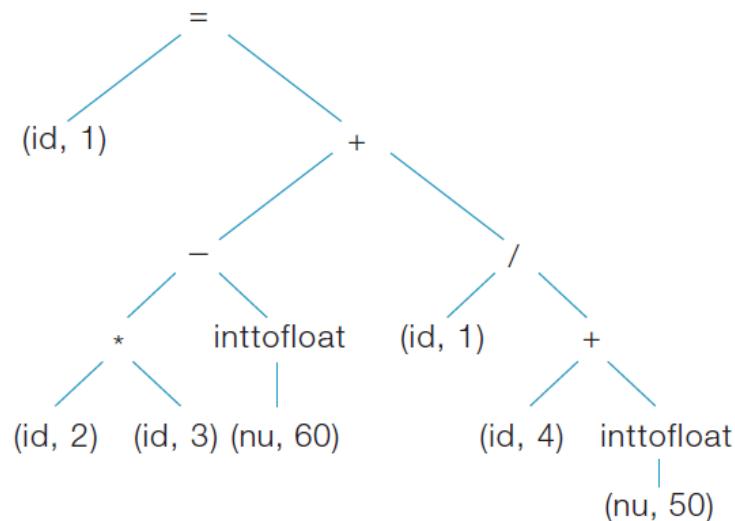
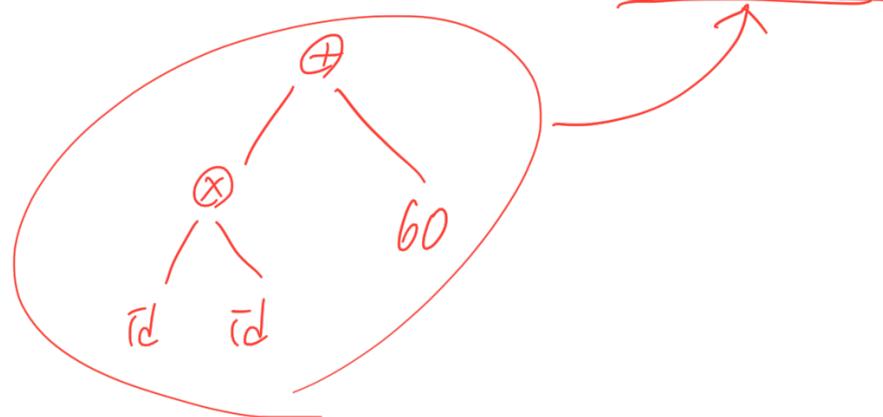


그림 2-9 [그림 2-8]에 대한 의미 분석 후의 구문 트리

2.1 컴파일러의 논리적 구조

▪ 중간코드생성(intermediate code generation) *Front-end 의 일정*

- 구문분석 단계에서 만들어진 구문 트리를 이용하여 코드를 생성하거나 한 문법규칙이 감축될 때마다 문법지시적 번역(syntax-directed translation)으로 이루어짐
- 중간코드 생성을 담당하는 도구를 중간코드 생성기(intermediate code generator)
- 중간 코드를 생성하기 위해서는 *three -address - code* 중간 언어가 필요
 - 중간언어 – 역폴란드식 표기(Fortran의 산술식, Basic의 interpreter), 폴란드식 표기(prefix)
- 중간 언어 중 3-주소 코드는 각 명령어마다 3개의 피연산자를 가진 $A = B \text{ op } C$ 형태의 명령어의 나열이다.



2.1 컴파일러의 논리적 구조

- 치환문 $ni = ba * po - 60 + ni / (abc + 50)$;에 대해 3-주소 코드(three-address code)를 이용한다면 다음과 같은 중간 코드가 생성된다.

temp1 = id2 * id3

temp2 = inttofloat(60)

temp3 = temp1 - temp2

temp4 = inttofloat(50)

temp5 = id4 + temp4

temp6 = id1 / temp5

temp7 = temp3 + temp6

id1 = temp7

- 여기서 각 명령어의 계산된 값을 담고 있을 임시 기억 장소인 temp1, temp2, temp3, temp4, temp5, temp6, temp7과 같은 임시 변수를 생성해야 한다.

2.1 컴퓨터의 논리적 구조

▪ 코드 최적화(code optimization)

- 같은 기능을 유지하면서 코드를 좀 더 효율적으로 만들어 코드 수행 시 기억 공간이나 수행시간을 절약하기 위한 단계
- 선택적인 단계로서 생략되는 경우도 있지만 요즈음에 와서는 RISC (Reduced Instruction Set Computer)와 같은 컴퓨터 구조의 특성을 활용하기 위해서 최적화 단계를 많이 사용하고 있다
- 최적화 방법은 최적화가 적용되는 프로그램의 영역에 따라서 지역 최적화 (local optimization), 전역 최적화(global optimization), 프로시저간 최적화 (inter-procedural optimization),
- 기능적인 측면으로서는 실행시간 최적화와 메모리 최적화로 분류,
- 최적화가 많이 이루어지는 부분에 따라서는 루프(loop 혹은 반복문) 최적화 와 단일문 최적화로 구분.
- 또한, 목적 기계의 의존성에 따라서 기계 독립 최적화(machine independent optimization)와 기계 종속 최적화(machine dependent optimization) 등이 있다.

Architecture front-end 최적화.

(ARM의 1dm)
→ ARM에서 지원하는
여러 개의 instruction을 합칠 수 있다

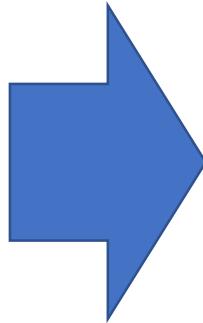
특징 고려 최적화.

gcc -O | 0번의 숫자가 올라갈수록
gcc -O2 | 최적화가 많이 됨
 : ⇒ 프로그램이 더 작아짐.

2.1 컴파일러의 논리적 구조

- 최적화를 수행하기 위해서는 주어진 중간 코드를 기본 블록(basic block)과 흐름 그래프(flow graph)가 필요하다.
- 치환문 $ni = ba * po - 60 + ni / (abc + 50);$ 에 대해 3-주소 코드로 생성된 중간 코드를 가지고 간략한 코드 최적화를 하면 [그림 2-10]과 같다.

```
temp1 = id2 * id3  
temp2 = inttofloat(60)  
temp3 = temp1 - temp2  
temp4 = inttofloat(50)  
temp5 = id4 + temp4  
temp6 = id1 / temp5  
temp7 = temp3 + temp6  
id1 = temp7
```



```
temp1 = id2 * id3  
temp3 = temp1 - 60.0  
temp5 = id4 + 50.0  
temp6 = id1 / temp5  
id1 = temp3 + temp6
```

그림 2-10 **최적화된 코드**

2.1 컴파일러의 논리적 구조

▪ 목적 코드 생성(code generation)

- 연산을 수행할 레지스터를 선택하거나 자료에 기억장소의 위치를 정해주며 실제로 목적 기계에 대한 코드를 생성하는 단계
- 레지스터 두 개(R_1, R_2)를 사용하는 경우의 예 :

LOAD	$R_1, id2$
LOAD	$R_2, id3$
MULT	R_2, R_1
SUBT	$R_2, \#60.0, R_2$
LOAD	$R_1, id4$
ADD	$R_1, \#50.0, R_1$
STORE	W_1, R_1
LOAD	$R_1, id1$
DIV	R_1, W_1, R_1
ADD	R_2, R_1
STORE	$id1, R_1$

target machine 의
Machine code :

종합 언어 코드의 경우에는 temporary register 를
많이 사용했다.

* 목적 코드의 경우 제한적인 레지스터를 사용해야 한
(register allocation)

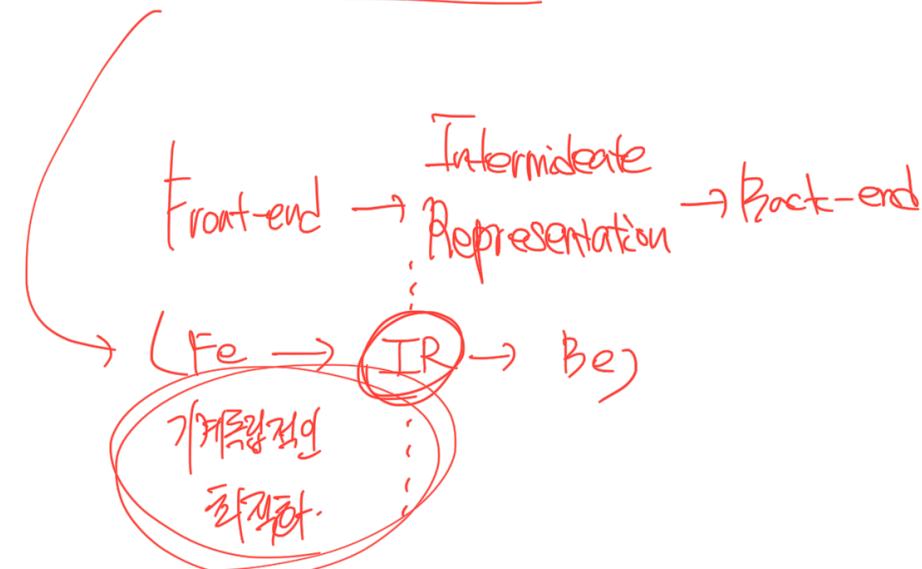
2.2 컴파일러의 물리적 구조 → 구현 단계에서 고려해야 할 사항.

▪ 어떻게 구현 할 것인가?

- One pass : 실행 속도가 빠르고, 효율성이 좋다. but. 새롭 language. architecture
가 아닌 때 따라 서로 만들어야 함.
- Two pass : 각 모듈 단위로 작성, 이식성, 기계독립적인 최적화 가능

▪ 컴파일러를 설계할 때 고려 사항 :

- 컴파일러의 논리적 구조
- 컴퓨터 자원
- 사용자의 요구 사항
- 개발하는 인적 자원



정리

■ 컴파일러의 논리적 구조

- 어휘 분석
- 구문 분석
- 의미 분석
- 코드 최적화 및 생성

(2차 2강)

각 단계별 암호화, 전제적인 의미 파악.

■ 컴파일러의 물리적 구조

- One-pass
- Two-pass