제 2 장

간단한 컴파일러의 구조

제 2 장 간단한 컴파일러의 구조

❖목적 :

- **1. 컴파일러의 논리적 구조**(컴파일러의 논리적 구조를 이해하고, 간단한 컴파일러의 예를 통하여 컴파일러의 전체 구조를 이해하기.)
- **2. 컴파일러의 물리적 구조**(컴파일러의 물리적 구조를 이해하기)

2.1 컴파일러의 논리적 구조

■ 영어를 한글로 번역

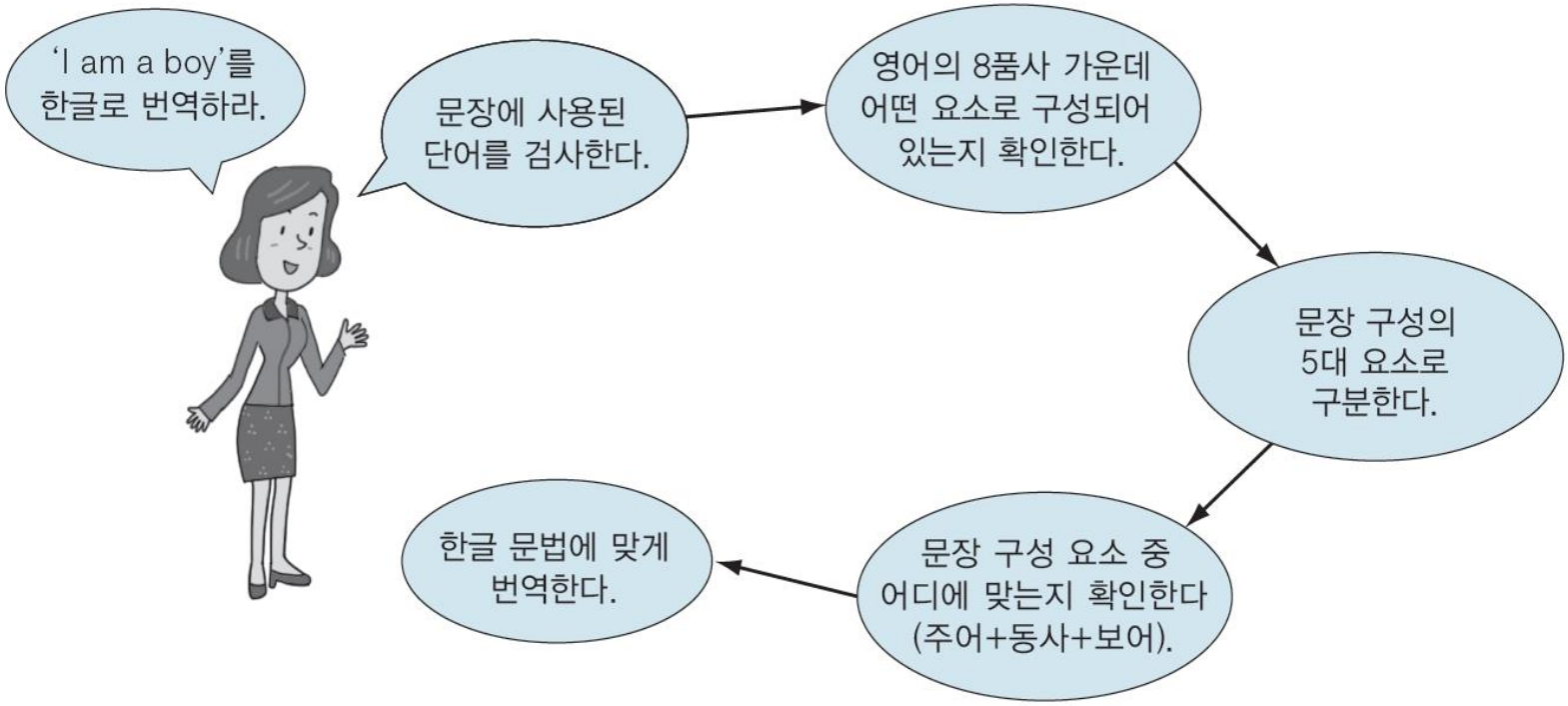


그림 2-1 영어를 한글로 번역

2.1 컴파일러의 논리적 구조

- 문장이 어떤 요소로 구성되어 있는지 파악하기 위해 문장에 사용된 단어를 검사.
  - ✓그래서 이 문장에는 I, am, a, boy라는 네 가지 단어가 사용된 것을 알아내는데 이를 어휘 분석이라 한다.
- 다음으로 I, am, a, boy가 각각 8품사인 명사, 대명사, 동사, 형용사, 부사, 전치사, 접속사, 감탄사 중 어디에 속하는지 확인하고, 문장의 5대 요소인 주어, 동사, 목적어, 보어, 수식어 등으로 구분.
  - ✓I am a boy는 대명사, 동사, 명사로 구성
  - ✓그리고 주어+동사, 주어+동사+보어, 주어+동사+목적어 등 문장의 형식을 검사하여 이 문장이 주어+동사+보어로 구성되었다는 것을 알아낸다. 이렇게 문장의 형식을 알아내는 것을 구문 분석이라 한다.
- 그 다음에는 단어 대 단어의 번역이 아니라 한글 문법에 맞게 번역해야 하는데 이를 의미 분석이라 한다.
- 그러고 나서 한글 단어 가운데 더 알맞은 단어로 번역하는 코드 최적화를 거쳐 마지막으로 한글로 번역하는 것이다.

2.1 컴파일러의 논리적 구조

- 컴파일러도 영어 문장을 한글로 번역하는 것과 비슷한 단계 거쳐서 번역
  - ✓C 언어를 기계어로 번역
  - ✓먼저 주어진 문장이 어떤 단어들을 포함하고 있는지 확인한다.
  - ✓C 언어에서 사용하는 의미 있는 단어를 '토큰(token)'이라고 한다.
  - ✓C 언어에서 어떤 토큰이 사용되었는지 구분하는 것인데, 이를 어휘 분석.
  - ✓두 번째로는 이러한 단어가 문장의 구성 요소 가운데 어디에 맞는지 확인해야 한다. 즉 문법을 보면서 토큰들이 문법에 맞는지 검사하는데, 컴파일러에서는 이를 구문 분석.
  - ✓세 번째로는 한국어의 의미를 검사하는 것, 즉 의미 분석을 한다. 하지만 형식 언어는 의미를 가지고 있지 않기 때문에 이 과정에서 주로 형(type)을 검사.
  - ✓의미 분석으로 생성된 코드의 실행 시간을 줄이거나 기억 장소를 줄이기 위한 코드 최적화
  - ✓마지막으로 목적 기계(target machine)의 특성을 고려하여 목적 코드를 생성.

2.1 컴파일러의 논리적 구조

- **컴파일러의 구조에 대한 견해 :**
  - 분석(Analysis)-합성(Synthesis) 모델 : 논리적인 구성 측면
  - 전단부(Front-end)-후단부(Back-end) : 기계 독립(machine independent) or 기계 종속(dependent), 구현측면
- **컴파일러의 논리적 구조**

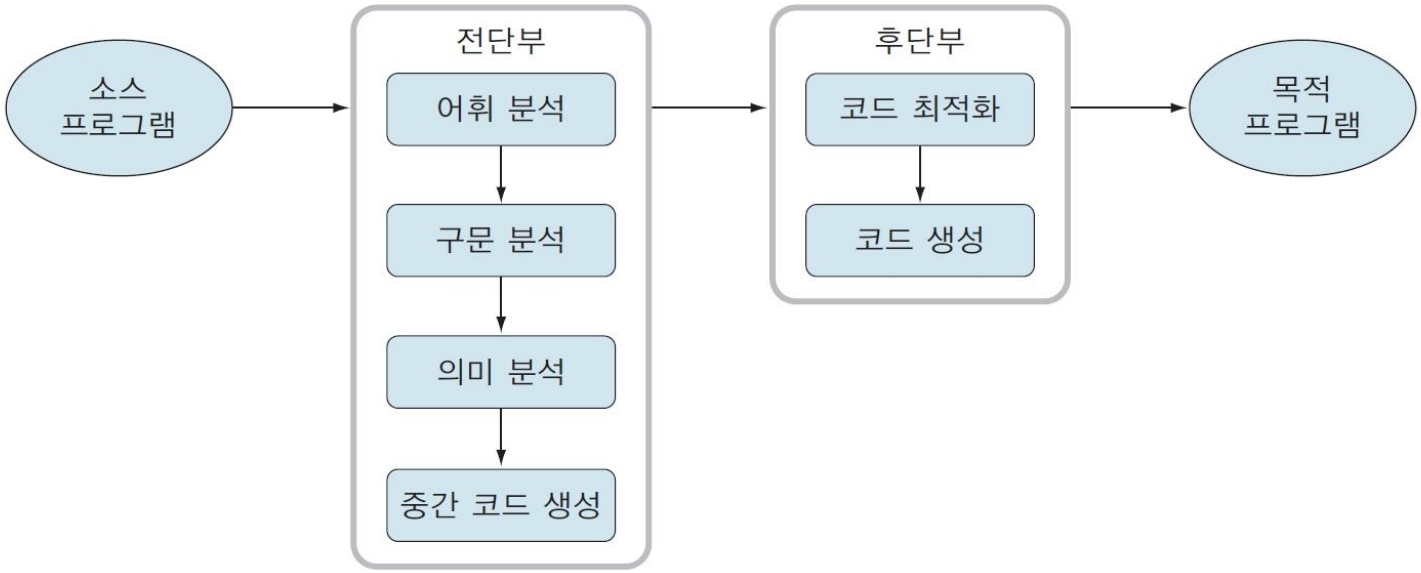


그림 2-2 컴파일러를 전단부와 후단부로 나눈 논리적 구조

2.1 컴파일러의 논리적 구조

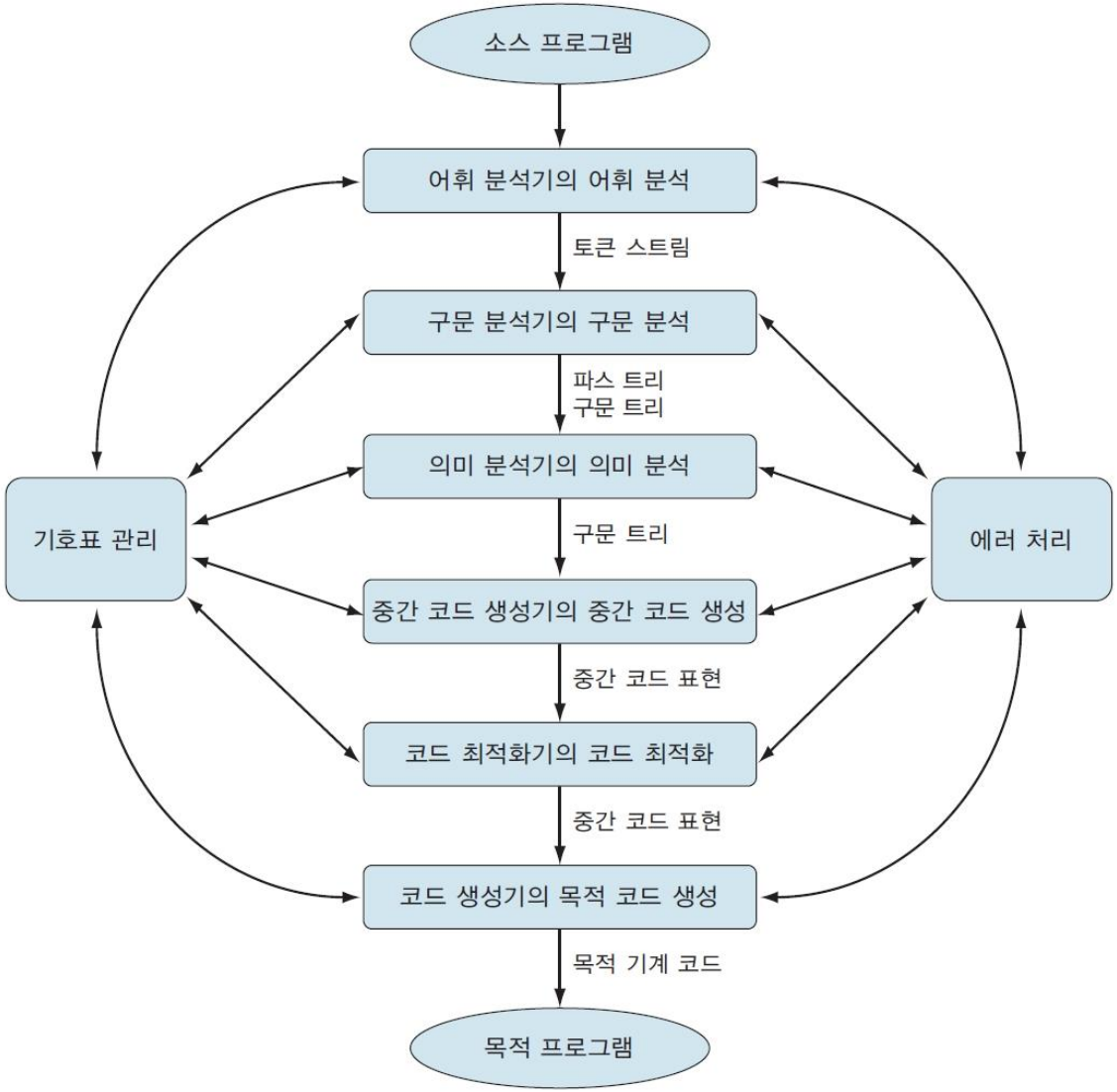


그림 2-3 컴파일러의 논리적 구조

2.1 컴파일러의 논리적 구조

■ 아주 간단한 C 문법

❶ ⟨Sub C⟩ ::= ⟨assign–st⟩
❷ ⟨assign–st⟩ ::= ⟨lhs⟩ = ⟨exp⟩;
❸ ⟨lhs⟩ ::= ⟨variable⟩
❹ ⟨exp⟩ ::= ⟨exp⟩+⟨term⟩ | ⟨exp⟩–⟨term⟩ | ⟨term⟩
❺ ⟨term⟩ ::= ⟨term⟩*⟨factor⟩ | ⟨term⟩/⟨factor⟩ | ⟨factor⟩
❻ ⟨factor⟩ ::= ⟨variable⟩ | ⟨number⟩ | (⟨exp⟩)
❼ ⟨variable⟩ ::= ⟨ident⟩
❽ ⟨ident⟩ ::= (⟨letter⟩ | _){⟨letter⟩ | ⟨digit⟩ | _}
❾ ⟨number⟩ ::= {⟨digit⟩}
❿ ⟨letter⟩ ::= a |⋯| z
⓫ ⟨digit⟩ ::= 0 | 1 |⋯| 9

그림 2–4 아주 간단한 C 문법

2.1 컴파일러의 논리적 구조

- 생성 규칙 ❶의 <Sub C>는 <assign-st>로 치환된다는 것을 의미한다. 생성 규칙 ❷의 <assign-st>는 <lhs>가 먼저 오고 =가 그 다음으로 오며 <exp>가 온 뒤 마지막으로 ;이 오는 문법이다.
- [그림 2-4]는 C 언어의 치환문을 표현하고 있으며, 산술식에는 산술 연산자 +, -, *, / 등 사칙 연산을 허용하는데 우선순위는 *와 /가 높고 그 다음이 +, -이며, 모든 연산자가 왼쪽 결합 법칙을 취함을 알 수 있다.
- 또한 산술식에 괄호를 사용할 수 있으며, 식별자는 첫 글자가 영문자나 언더바로 시작하고 두 번째 글자 부터는 영문자, 숫자, 언더바가 올 수 있으며 길이의 제한이 없다. 그리고 숫자는 0과 양의 정수를 사용할 수 있으며, 영문자는 소문자 a부터 z까지 사용 가능하다.

2.1 컴파일러의 논리적 구조

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

- [풀이]

- <Sub C> ⇒ <assign-st> (∵ 생성 규칙 ❶ 적용)
- ⇒ <lhs> = <exp>; (∵ 생성 규칙 ❷ 적용)
- ⇒ <variable> = <exp>; (∵ 생성 규칙 ❸ 적용)
- ⇒ <ident> = <exp>; (∵ 생성 규칙 ❼ 적용)
- ⇒ (<letter>|_){<letter>|<digit>|_} = <exp>; (∵ 생성 규칙 ❽ 적용)
- ⇒ n{<letter>|<digit>|_} = <exp>; (∵ 생성 규칙 ❿ 적용)
- ⇒ ni = <exp>; (∵ 생성 규칙 ❿ 적용)
- ⇒ ni = <term>; (∵ 생성 규칙 ❹ 적용)

2.1 컴파일러의 논리적 구조

- ⇒ ni = <factor>; (∵ 생성 규칙 ❺ 적용)
- ⇒ ni = (<exp>); (∵ 생성 규칙 ❻ 적용)
- ⇒ ni = (<exp>-<term>); (∵ 생성 규칙 ❹ 적용)
- ⇒ ni = (<term>-<term>); (∵ 생성 규칙 ❹ 적용)
- ⇒ ni = (<factor>-<term>); (∵ 생성 규칙 ❺ 적용)
- ⇒ ni = (<variable>-<term>); (∵ 생성 규칙 ❻ 적용)
- ⇒ ni = (<ident>-<term>); (∵ 생성 규칙 ❼ 적용)
- ⇒ ni = ((<letter>|_){<letter>|<digit>|_}-<term>); (∵ 생성 규칙 ❽ 적용)
- ⇒ ni = (p{<letter>|<digit>|_}-<term>); (∵ 생성 규칙 ❿ 적용)
- ⇒ ni = (po - <term>); (∵ 생성 규칙 ❿ 적용)
- ⇒ ni = (po - <factor>); (∵ 생성 규칙 ❺ 적용)
- ⇒ ni = (po - <number>); (∵ 생성 규칙 ❻ 적용)
- ⇒ ni = (po - {<digit>}); (∵ 생성 규칙 ❾ 적용)
- ⇒ ni = (po - 60); (∵ 생성 규칙 11 적용)

- 치환문 ni = (po - 60);이 주어진 문법에 맞는 문장이다.

2.1 컴파일러의 논리적 구조

- **[예제 2-2]** 치환문 ni = ba * po - 60 + ni / (abc + 50);이 [그림 2-4]의 문법에 맞는 문장인지 알아보자. (설명은 생략)
  - [풀이]
    - <Sub C> ⇒ <assign-st> (∵ 생성 규칙 ❶ 적용)
    - ⇒ <lhs> = <exp>; (∵ 생성 규칙 ❷ 적용)
    - ⇒ <variable> = <exp>; (∵ 생성 규칙 ❸ 적용)
    - ⇒ <ident> = <exp>; (∵ 생성 규칙 ❼ 적용)
    - ⇒ (<letter>|_){<letter>|<digit>|_} = <exp>; (∵ 생성 규칙 ❽ 적용)
    - ⇒ n{<letter>|<digit>|_} = <exp>; (∵ 생성 규칙 ❿ 적용)
    - ⇒ ni = <exp>; (∵ 생성 규칙 ❿ 적용)
    - ⇒ ni = <exp>+<term>; (∵ 생성 규칙 ❹ 적용)
    - ⇒ ni = <exp>-<term>+<term>; (∵ 생성 규칙 ❹ 적용)
    - ⇒ ni = <term>-<term>+<term>; (∵ 생성 규칙 ❹ 적용)
    - ⇒ ni = <term>*<factor>-<term>+<term>; (∵ 생성 규칙 ❹ 적용)
    - ⇒ ni = <factor>*<factor>-<term>+<term>; (∵ 생성 규칙 ❺ 적용)
    - ⇒ ni = <variable>*<factor>-<term>+<term>; (∵ 생성 규칙 ❻ 적용)
    - ⇒ ni = <ident>*<factor>-<term>+<term>; (∵ 생성 규칙 ❼ 적용)
    - ⇒ ni = (<letter>|_){<letter>|<digit>|_}*<factor>-<term>+<term>; (∵ 생성 규칙❽ 적용)

2.1 컴파일러의 논리적 구조

- ⇒ ni = b{<letter>|<digit>|_}*<factor>-<term>+<term>; (∵ 생성 규칙⑩ 적용)
- ⇒ ni = ba * <factor>-<term>+<term>; (∵ 생성 규칙 ⑩ 적용)
- ⇒ ni = ba * <variable>-<term>+<term>; (∵ 생성 규칙 ❻ 적용)
- ⇒ ni = ba * <ident>-<term>+<term>; (∵ 생성 규칙 ❼ 적용)
- ⇒ ni = ba * (<letter>|_){<letter>|<digit>|_}-<term>+<term>; (∵ 생성 규칙❽ 적용)
- ⇒ ni = ba * p{<letter>|<digit>|_}-<term>+<term>; (∵ 생성 규칙⑩ 적용)
- ⇒ ni = ba * po - <term>+<term>; (∵ 생성 규칙 ⑩ 적용)
- ⇒ ni = ba * po - <factor>+<term>; (∵ 생성 규칙 ❺ 적용)
- ⇒ ni = ba * po - <number>+<term>; (∵ 생성 규칙 ❻ 적용)
- ⇒ ni = ba * po - {<digit>}+<term>; (∵ 생성 규칙 ❾ 적용)
- ⇒ ni = ba * po - 60 + <term>; (∵ 생성 규칙 11 적용)
- ⇒ ni = ba * po - 60 + <term>/<factor>; (∵ 생성 규칙 ❺ 적용)
- ⇒ ni = ba * po - 60 + <factor>/<factor>; (∵ 생성 규칙 ❺ 적용)
- ⇒ ni = ba * po - 60 + <variable>/<factor>; (∵ 생성 규칙 ❻ 적용)
- ⇒ ni = ba * po - 60 + <ident>/<factor>; (∵ 생성 규칙 ❼ 적용)
- ⇒ ni = ba * po - 60 + (<letter>|_){<letter>|<digit>|_}/<factor>; (∵ 생성 규칙❽ 적용)
- ⇒ ni = ba * po - 60 + n{<letter>|<digit>|_}/<factor>; (∵ 생성 규칙⑩ 적용)
- ⇒ ni = ba * po - 60 + ni /<factor>; (∵ 생성 규칙 ⑩ 적용)
- ⇒ ni = ba * po - 60 + ni /(<exp>+<term>); (∵ 생성 규칙 ❹ 적용)

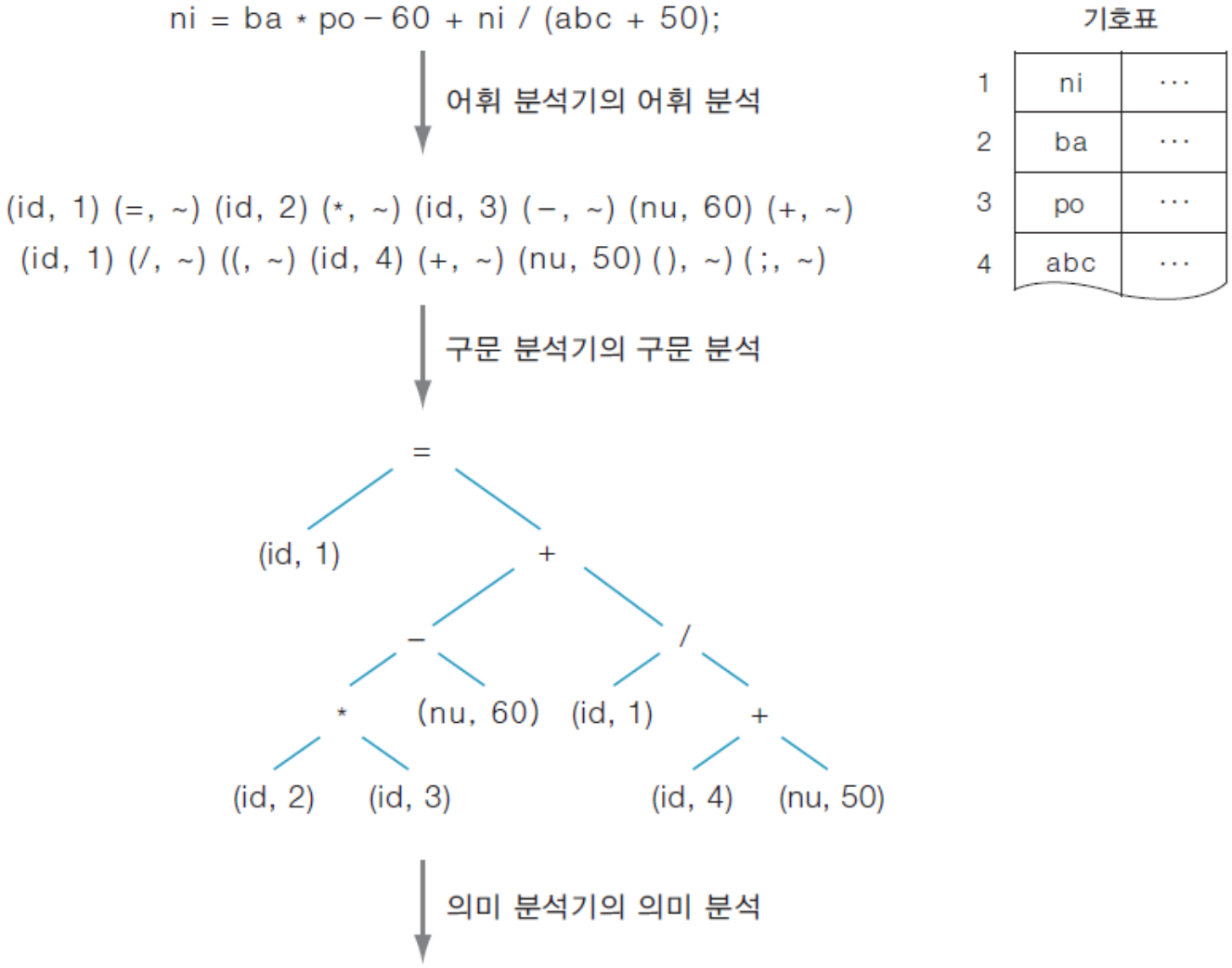
2.1 컴파일러의 논리적 구조

- ⇒ ni = ba * po - 60 + ni /(<term>+<term>); (∵ 생성 규칙 ❹ 적용)
- ⇒ ni = ba * po - 60 + ni /(<factor>+<term>); (∵ 생성 규칙 ❺ 적용)
- ⇒ ni = ba * po - 60 + ni /(<variable>+<term>); (∵ 생성 규칙 ❻ 적용)
- ⇒ ni = ba * po - 60 + ni /(<ident>+<term>); (∵ 생성 규칙 ❼ 적용)
- ⇒ ni = ba * po - 60 + ni /((<letter>|_){<letter>|<digit>|_}+<term>); (∵ 생성 규칙 ❽ 적용)
- ⇒ ni = ba * po - 60 + ni /(a{<letter>|<digit>|_}+<term>); (∵ 생성 규칙 ❿ 적용)
- ⇒ ni = ba * po - 60 + ni /(abc+<term>); (∵ 생성 규칙 ❿ 적용)
- ⇒ ni = ba * po - 60 + ni /(abc+<factor>); (∵ 생성 규칙 ❺ 적용)
- ⇒ ni = ba * po - 60 + ni /(abc+<number>); (∵ 생성 규칙 ❻ 적용)
- ⇒ ni = ba * po - 60 + ni /(abc+{<digit>}); (∵ 생성 규칙 ❾ 적용)
- ⇒ ni = ba * po - 60 + ni /(abc+50); (∵ 생성 규칙 11 적용)

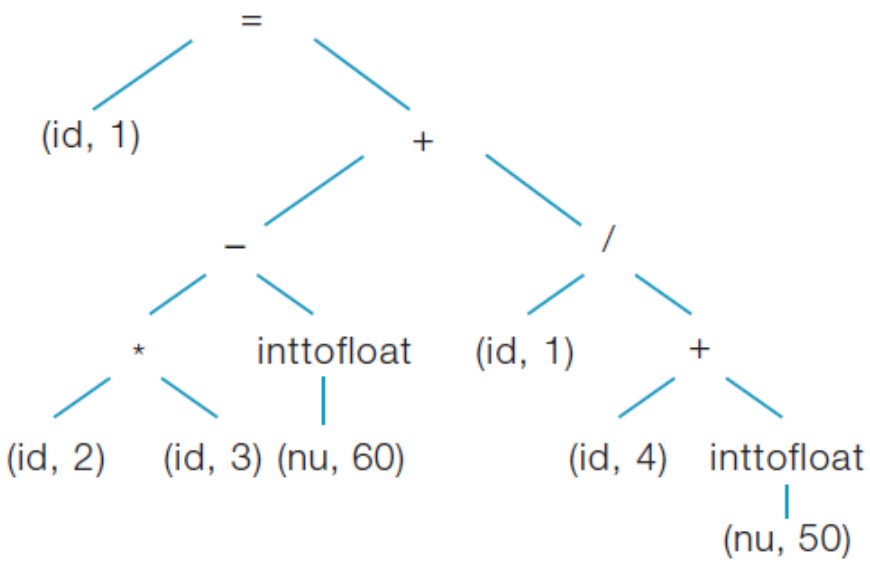
■ 이와 같이 좌단 유도에 의해 치환문 ni = ba * po - 60 + ni / (abc + 50);은 [그림 2-4]의 문법에 맞는 문장임을 알 수 있다.

2.1 컴파일러의 논리적 구조

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



2.1 컴파일러의 논리적 구조



중간 코드 생성기의 중간 코드 생성

```
t1 = id2 * id3
t2 = inttofloat(60)
t3 = t1 − t2
t4 = inttofloat(50)
t5 = id4 + t4
t6 = id1 / t5
t7 = t3 + t6
id1 = t7
```

코드 최적화기의 코드 최적화

```
t1 = id2 * id3
t3 = t1 − 60.0
t5 = id4 + 50.0
t6 = id1 / t5
id1 = t3 + t6
```

코드 생성기의 목적 코드 생성

```
LOAD   R1, id2
LOAD   R2, id3
MULT   R1, R2
SUBT   R2, #60.0, R2
LOAD   R1, id4
ADD    R1, #50.0, R1
STORE  W1, R1
LOAD   R1, id1
DIV    R1, W1, R1
ADD    R2, R1
STORE  id1, R1
```

그림 2-5 치환문의 컴파일링 과정

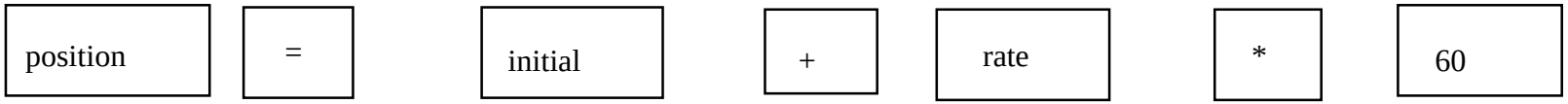
2.1 컴파일러의 논리적 구조

- LOAD R0, y //y의 값을 레지스터 R0에 저장
- OP R1, R2, R3 // OP는 연산 명령어이고 R3 ← R1 + R2 이다. 단지 OP R1, R2, R2 인 경우는 OP R1, R2로 표현
- STORE x, R0 // x = R0

2.1 컴파일러의 논리적 구조

어휘 분석(lexical analysis, scanning)

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



■ 토큰의 종류

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

토큰 테이블 작성

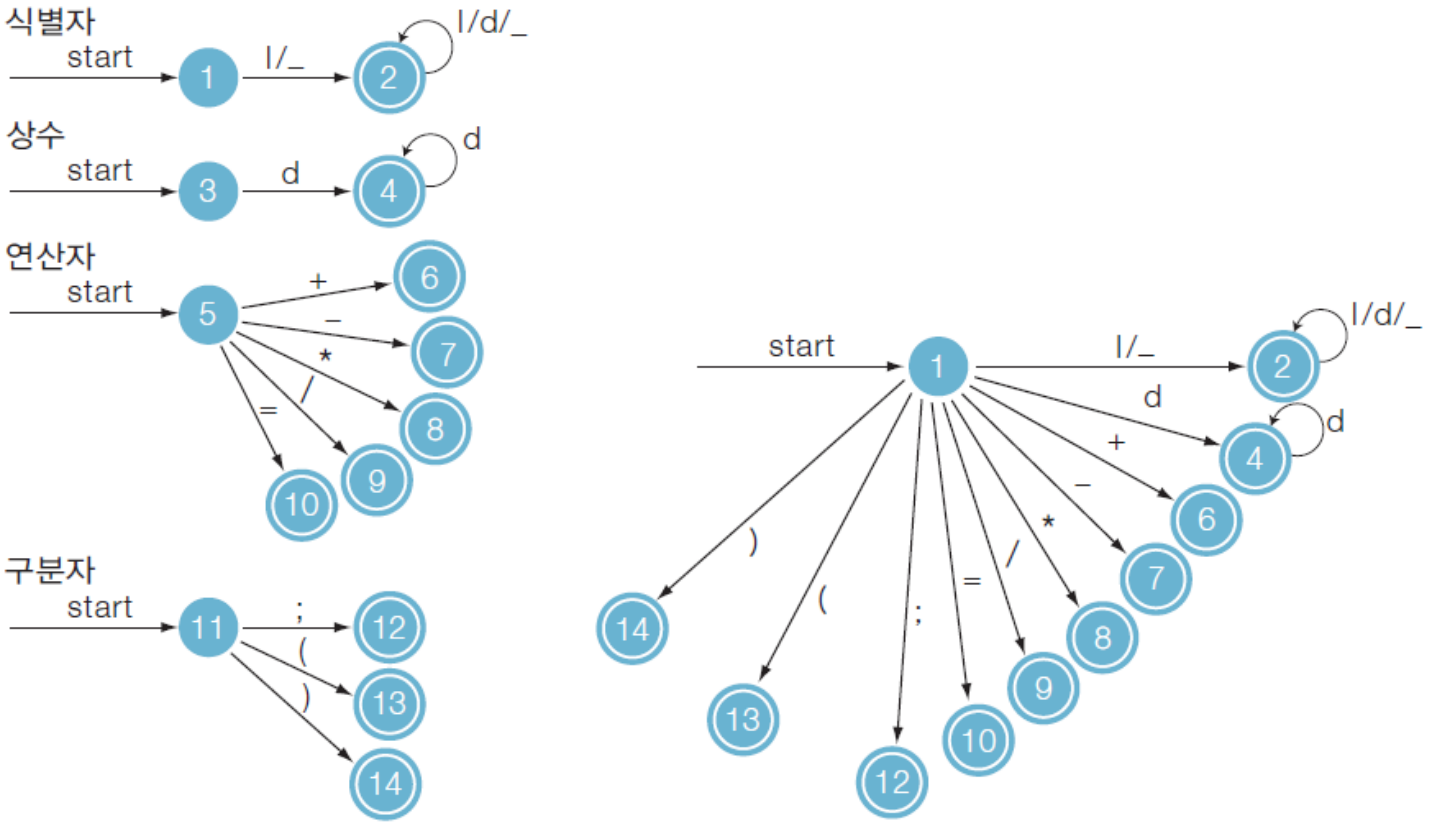
- 문법을 보고 어떤 토큰 들이 사용되었는지 표로 만듬

2.1 컴파일러의 논리적 구조

- [그림 2.4]의 문법에서 예약어가 허용되는지를 확인한다, [그림 2.4]에서는 예약어를 사용하지 않는다. 상수를 보면, 0이나 양의 정수를 허용한다. 연산자는 4칙 연산자인 +, -, *, / 를 사용하고 치환연산자인 =를 허용한다. 식별자도 허용하고 첫 자는 영문자나 언더바로 시작하고 두 번째 자 부터는 영문자나 숫자, 언더바를 사용하고 길이에는 제한이 없다. 구분자로는 세미콜론만이 사용된다. 이들을 정규 문법과 정규 표현으로 나타내면 다음과 같다.
  - 1) 식별자 : 정규 문법
  - S → lA | _A
  - A → lA | dA | _A | ε 로부터 정규표현으로 나타내기 위해서 정규 표현 방정식은 다음과 같다.
  - S = (l + _)A
  - A = lA | dA | _A | ε = (l + d +_)A + ε = (l + d +_)*
  - 이것으로부터 해를 구하면
  - S = (l + _)A = (l + _)(l + d +_)*
  - 2) 상수 : S= $(d)^{+}$
  - 3) 연산자 : S = + | - | * | / | '='
  - 4) 구분자 : ;, (,)

2.1 컴파일러의 논리적 구조

- 이들을 받아들이는 유한 오토마타는 다음과 같다.(같은 예제가 4장과 13장)



(a) 각각의 토큰에 대한 어휘 분석기 (b) 전체 토큰에 대한 어휘 분석기

그림 2-6 [그림 2-4]에 대한 어휘 분석기

2.1 컴파일러의 논리적 구조

- 토큰에 대한 표현은 (토큰 번호, 토큰 값)의 순서쌍으로 표현

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

- 주석(comment)도 이 과정에서 처리

2.1 컴파일러의 논리적 구조

- **구문분석(syntax-analysis, parsing)**
  - 어휘 분석 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 토큰들을 터미널 노드(terminal node)로 하는 트리(tree)

2.1 컴파일러의 논리적 구조

- **파스 트리의 그림을 간단하게 표현하기 위하여 [그림 2-4]의 그림에 사용된 이름을 다음과 같이 간략하게 사용한다.**
  - ✓assign-st => as, lhs => lh, exp => ex, term => te, factor => fa, variable => va, ident => id
  - ✓또한, 파싱표에 사용하기 위해 생성 규칙 번호를 다음과 같이 정의한다.
    - ♣ 0. <S'> ::= <Sub C>
    - ♣ 1. <Sub C> ::= <as>
    - ♣ 2. <as> ::= <lh>=<ex>;
    - ♣ 3. <lh> ::= <va>
    - ♣ 4. <ex> ::= <ex> + <te>
    - ♣ 5. <ex> ::= <ex> - <te>
    - ♣ 6. <ex> ::= <te>
    - ♣ 7. <te> ::= <te> * <fa>
    - ♣ 8. <te> ::= <te> / <fa>
    - ♣ 9. <te> ::= <fa>
    - ♣ 10. <fa> ::= <va>
    - ♣ 11. <fa> ::= <nu>
    - ♣ 12. <fa> ::= (<ex>)
    - ♣ 13. <va> ::= <id>

2.1 컴파일러의 논리적 구조

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

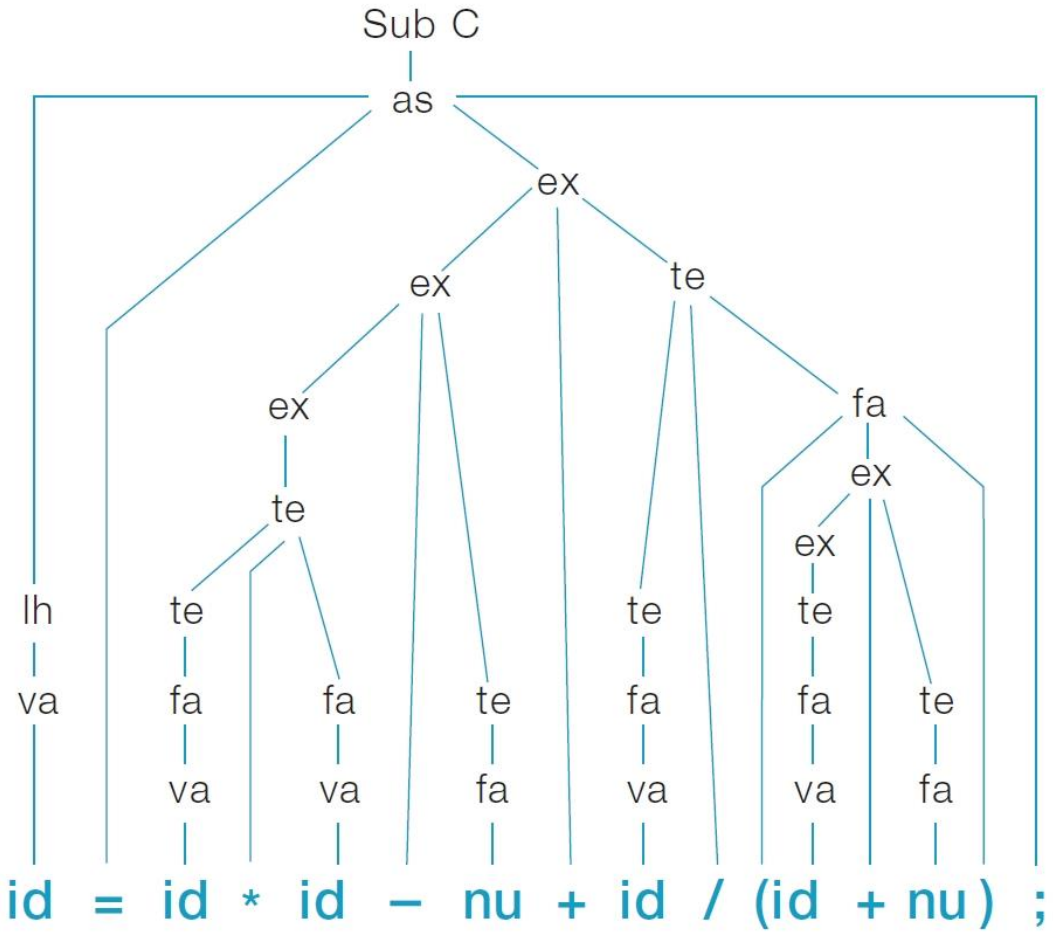


그림 2–7 치환문 ni = ba * po – 60 + ni / (abc + 50);의 파스 트리

2.1 컴파일러의 논리적 구조

- 파스 트리에 대한 구문 트리

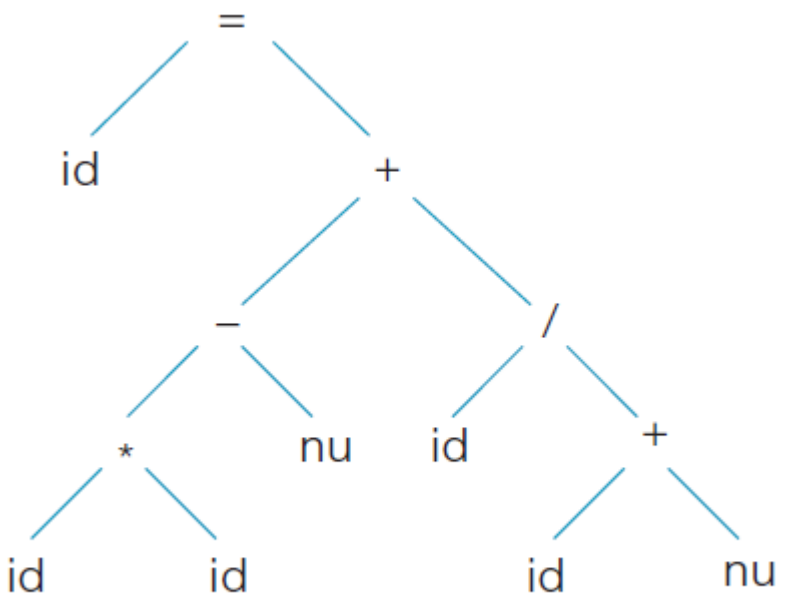


그림 2-8 [그림 2-7]의 파스 트리에 대한 구문 트리

2.1 컴파일러의 논리적 구조

- [그림 2-4]의 문법에 대한 SLR 파싱표가 [표 2-1]과 같을 때 이 파싱표를 가지고 구문 분석을 할 수 있다.

- 교재 53 페이지 표 2-1 참조

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

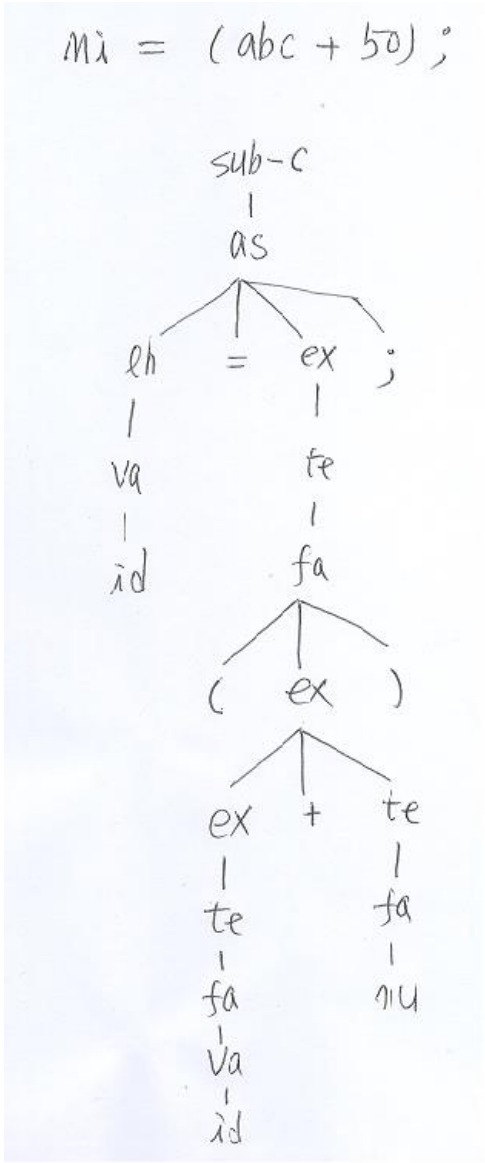
단계	스택	입력 기호	구문 분석 내용
40	0lh3=11ex18+12te19/15	(id+nu);$	shift 9
41	0lh3=11ex18+12te19/15(9	id+nu);$	shift 10
42	0lh3=11ex18+12te19/15(9id10	+nu);$	reduce 13
43	0lh3=11ex18+12te19/15(9va	+nu);$	goto 17
44	0lh3=11ex18+12te19/15(9va17	+nu);$	reduce 10
45	0lh3=11ex18+12te19/15(9fa	+nu);$	goto 7
46	0lh3=11ex18+12te19/15(9fa7	+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

- **[표 2-2]의 파싱에서 주어진 문장에 대해 수락되었으므로 치환문 ni = ba * po - 60 + ni / (abc + 50);은 주어진 문법에 맞는 문장임을 알 수 있다.**

2.1 컴파일러의 논리적 구조

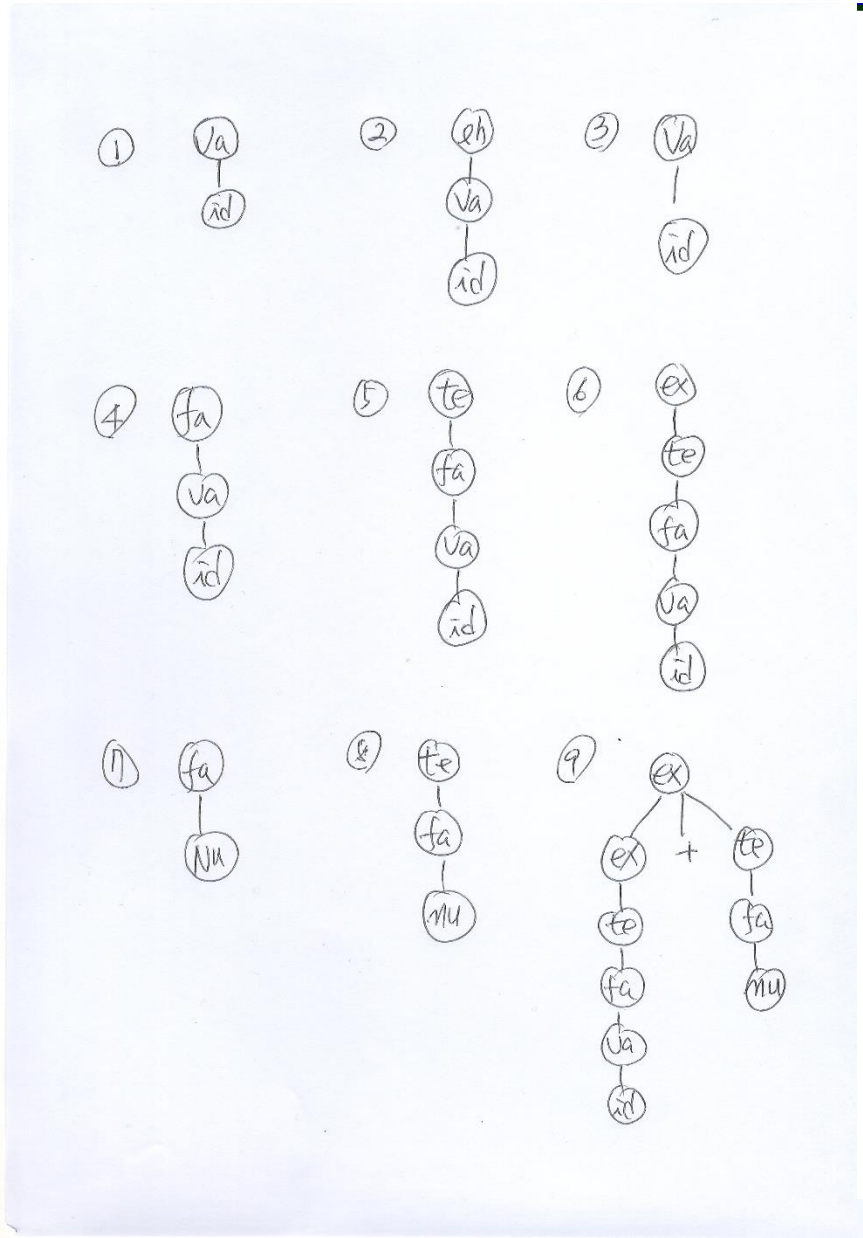
- 

2.1 컴파일러의 논리적 구조

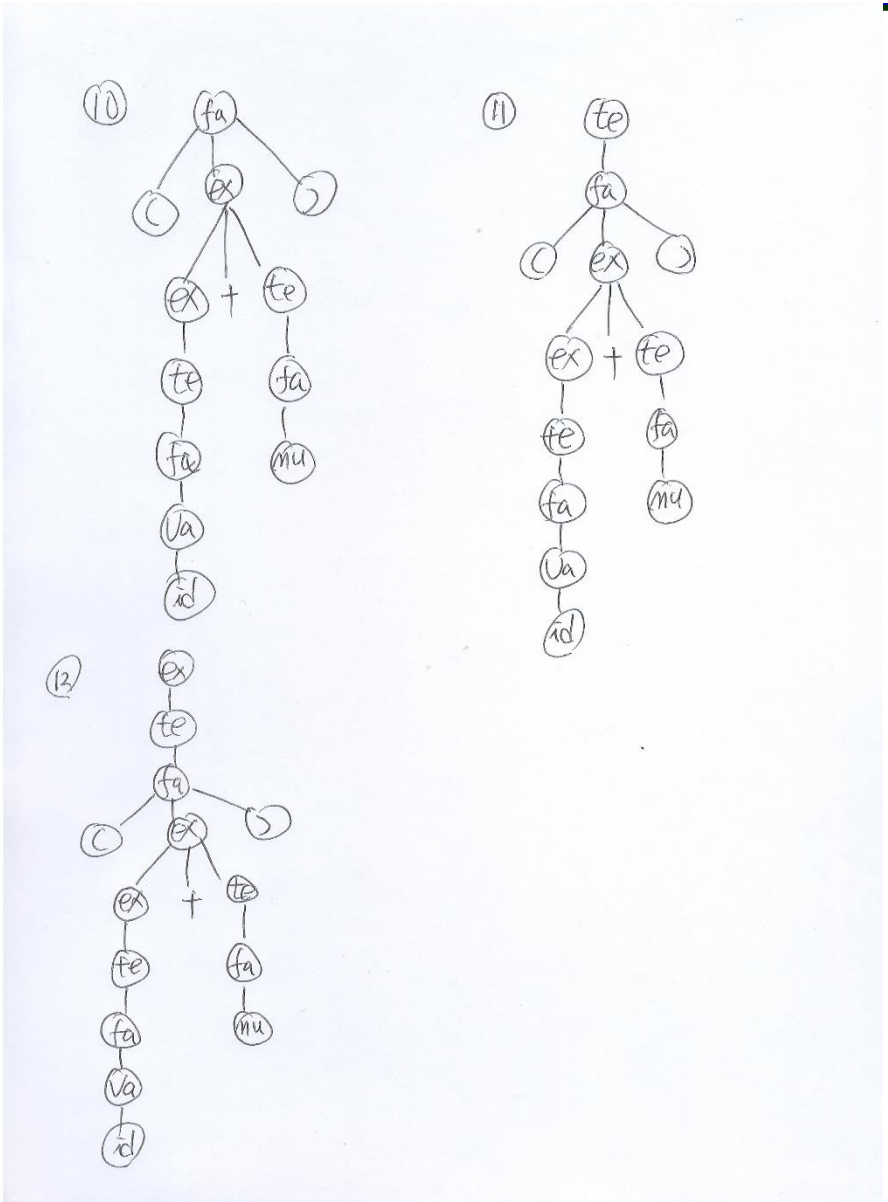
- id = (id + nu) ;

stack	input	Parse tree
0	id=(id+nu);$	
0 id 10	=(id+nu);$	①
0 va 4	=(id+nu);$	②
0 lh 3	=(id+nu);$	
0 lh 3 = 11	(id+nu);$	
0 lh 3 = 11 (9	id+nu);$	
0 lh 3 = 11 (9 id 10	+nu);$	③
0 lh 3 = 11 (9 va 17	+nu);$	④
0 lh 3 = 11 (9 fa 7	+nu);$	⑤
0 lh 3 = 11 (9 te 6	+nu);$	⑥
0 lh 3 = 11 (9 ex 16	+nu);$	
0 lh 3 = 11 (9 ex 16 + 12	nu);$	
0 lh 3 = 11 (9 ex 16 + 12 – nu 8	);$	⑦
0 lh 3 = 11 (9 ex 16 + 12 – fa 7	);$	⑧
0 lh 3 = 11 (9 ex 16 + 12 – te 19	);$	⑨
0 lh 3 = 11 (9 ex 16	);$	
0 lh 3 = 11 (9 ex 16) 23	;$	⑩
0 lh 3 = 11 fa 7	;$	⑪
0 lh 3 = 11 te 6	;$	⑫
0 lh 3 = 11 ex 18	;$	
0 lh 3 = 11 ex 18 ; 24	$	⑬
0 as 2	$	⑭
0 sub-c 1	$	accept

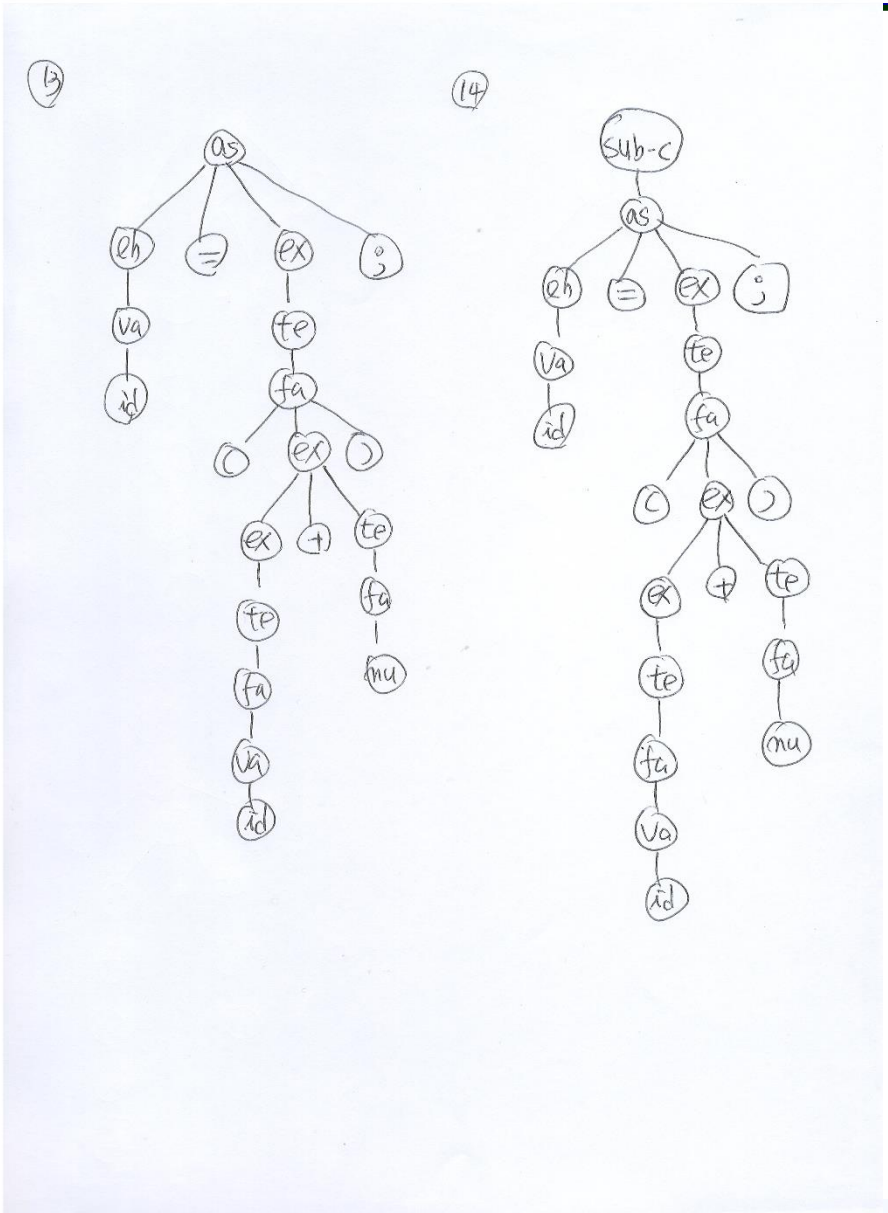
2.1 컴파일러의 논리적 구조



2.1 컴파일러의 논리적 구조



2.1 컴파일러의 논리적 구조



2.1 컴파일러의 논리적 구조

■ 의미 분석(semantic-analysis)

- 원시 프로그램의 의미적 오류를 검사하고 계속되는 코드 생성 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)
  - ✓각 연산자들이 원시 프로그램 규칙에 의해서 허용된 피연산자를 가졌는가를 검사(일반적 연산(generic operation)과 형 고정 연산(Type specific operation))
  - ✓정수와 실수의 일반적 연산을 허용하는데 이때의 의미 분석기는 연산을 수행하기 전에 정수를 실수로 바꾸어 주는 작업.(언어정의시간)
  - ✓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

- **[예 2-3]** 어떤 언어가 일반적 연산을 허용하고 정수형인 int와 실수형인 float가 있는 경우, 언어 정의 시간에 정수형을 실수형으로 변환하여 계산한다고 가정하고, 치환문 ni = ba * po - 60 + ni / (abc + 50);에서 나타난 식별자 ni, ba, po, ni, abc는 모두 실수형이라 고 가정한다. 이때 [그림 2-8]의 구문 트리를 가지고 의미 분석을 해보자.
- **[풀이]** 의미 분석하면 [그림 2.9]과 같다. 여기서 inttofloat는 정수형을 실수형으로 변환하는 함수.

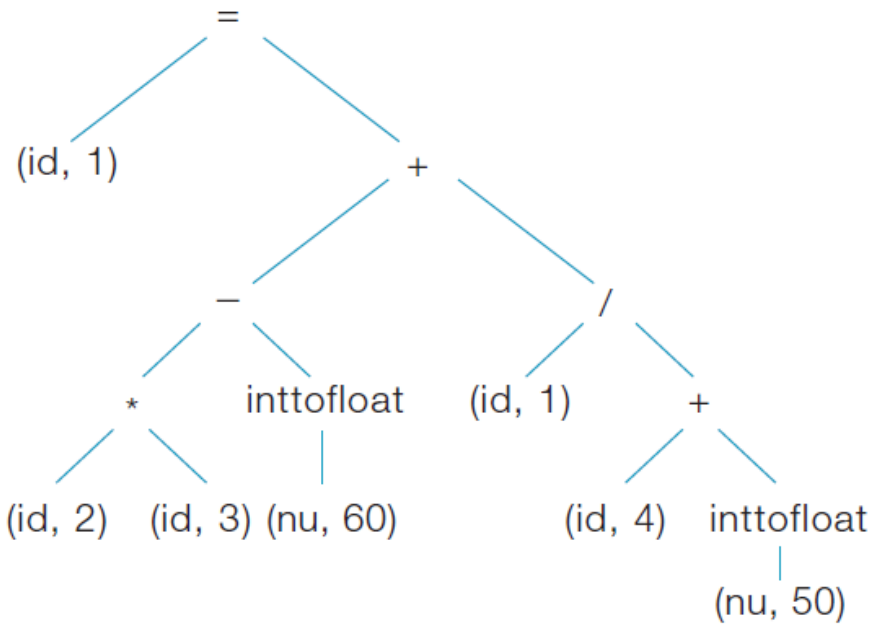


그림 2–9 [그림 2–8]에 대한 의미 분석 후의 구문 트리

2.1 컴파일러의 논리적 구조

중간코드생성(intermediate code generation)

- 구문분석 단계에서 만들어진 구문 트리를 이용하여 코드를 생성하거나 한 문법 규칙이 감축될 때마다 문법지시적 번역(syntax-directed translation)으로 이루어짐
- 중간 코드 생성을 담당하는 도구를 중간 코드 생성기(intermediate code generator)
- 중간 코드를 생성하기 위해서는 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 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.
- 8장에서 중간 언어와 중간 코드 생성에 대해 자세히 설명할 것이다.

2.1 컴파일러의 논리적 구조

코드 최적화(code optimization)

- 같은 기능을 유지 하면서 코드를 좀 더 효율적으로 만들어 코드 수행 시 기억 공간 이나 수행 시간을 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 실행 시간 최적화와 메모리 최적화로 분류, 최적화가 많이 이루어지는 부분에 따라서는 루프(loop 혹은 반복문) 최적화와 단일문 최적화로 구분. 또한, 목적 기계의 의존성에 따라서 기계 독립 최적화(machine independent optimization)와 기계 종속 최적화(machine dependent optimization) 등이 있다.
- 방법이 많기 때문에 cost effective 한 방법 찾음

2.1 컴파일러의 논리적 구조

지역 최적화

- 부분적인 관점에서 일련의 비효율적인 코드들을 구분해내고 좀더 효율적인 코드로 수정하는 방법으로 코드가 분기해 나가거나 분기해 들어오는 부분이 없는 기본 블록(basic block)내에서 최적화를 수행하므로 상대적으로 쉽게 수행할 수 있다. 기본 블록은 어떠한 제어 흐름도 가지지 않기 때문에 분석이 거의 필요 없다. 이 방법에는 공통 부분식 제거(common subexpression elimination), 복사 전파(copy propagation), 죽은 코드 제거(dead-code elimination), 상수 폴딩, 대수학적 간소화 등.

전역 최적화

- 기본 블록을 넘어서지만 하나의 프로시저 내에서 일련의 비효율적인 코드들을 구분해 내고 좀 더 효율적인 코드로 만드는 방법이다. 이 방법을 적용하기 위해서는 지역 최적화보다 더 많은 정보와 많은 비용을 요구해서 수행하기는 어렵지만, 지역 최적화보다 효과가 훨씬 좋다. 전역 최적화는 일반적으로 자료 흐름 분석(data flow analysis)이라는 기법을 사용하는데, 자료 흐름 분석은 프로그램 안에서 사용되는 변수의 경로(path)에 관한 정보를 수집해서 처리하는 과정이다. 이 방법에는 전역적 공통 부분식 제거, 상수 폴딩, 도달될 수 없는 코드의 제거 등.

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
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 기억 장소의 위치를 정해 주며 실제로 목적 기계에 대한 코드를 생성하는 단계
- 레지스터 두 개 (R_1, R_2)를 사용하는 경우의 예 :

```
LOAD    R1, id2
LOAD    R2, id3
MULT    R2, R1
SUBT    R2, #60.0, R2
LOAD    R1, id4
ADD     R1, #50.0, R1
STORE   W1, R1
LOAD    R1, id1
DIV     R1, W1, R1
ADD     R2, R1
STORE   id1, R1
```

2.2 컴파일러의 물리적 구조

- **어떻게 구현 할 것인가?**
  - One pass : 실행 속도가 빠르고, 효율성이 좋다.
  - 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