Chapter 02

간단한 컴파일러의 구조

Preview

이 장에서는 컴파일러의 전체적인 구조를 다룬다. 컴파일러의 구조는 일반적으로 논리적 구조와 물리적 구조로 나눌 수 있다. 먼저 컴파일러가 이론적으로 어떻게 구성되는지 논리적 구조에 대해 예를 통해 살펴본 다음, 컴파일러를 실제 제작할 때 필요한 물리적 구조에 대해 간단히 살펴본다.

01 컴파일러의 논리적 구조

컴파일러에 대한 내용을 전혀 다루지 않은 지금 예를 통해 컴파일러의 논리적 구조를 설명하면 매우 어려울 수 있다. 하지만 먼저 전체적인 개념을 알고 각각에 대해 살펴보는 것이 도움이 되리라 생각하여 간단하게 전체적인 개념을 다룬다.

1. 개요

컴파일러의 논리적 구조logical organization를 설명하기 위해 'I am a boy'라는 영어 문장을 살펴보자. 우리는 '나는 소년이다'라고 직관적으로 해석할 수 있지만, 컴퓨터는 [그림 2-1]과 같은 과정을 거쳐서 한국어로 번역할 것이다.

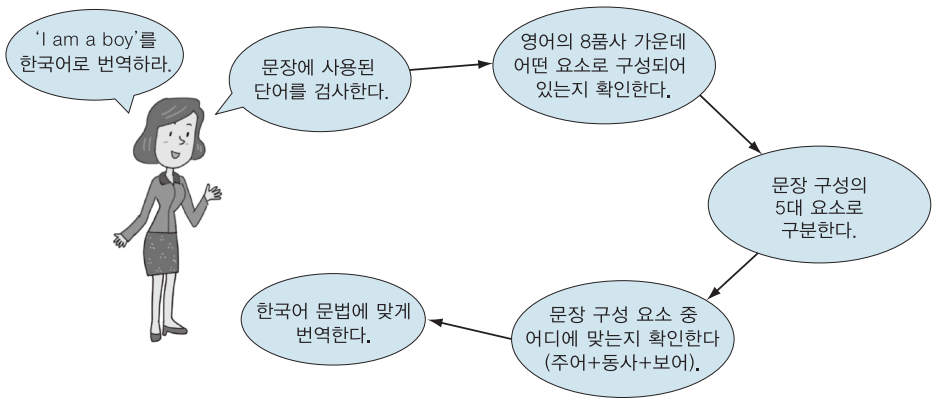


그림 2-1 **영어를 한국어로 번역**

먼저 문장이 어떤 요소로 구성되어 있는지 파악하기 위해 문장에 사용된 단어를 검사할 것이다. 그래서 이 문장에는 I, am, a, boy라는 네 가지 단어가 사용되었음을 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 구분할 것이다. 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. 그다음에는 단어 대 단어의 번역이 아니라 한국어 문법에 맞게 번역해야 하는데 이를 의미 분석이라 한다. 그리고 나서 한국어 단어 가운데 더 알맞은 단어로 번역하는 코드 최적화를 거쳐 마지막으로 한국어로

번역하는 것이다.

컴파일러도 이처럼 영어 문장을 한국어로 번역하는 것과 비슷한 단계를 거쳐서 번역을 하게 된다. C 언어를 기계어로 번역하는 데에도 유사한 방식을 따른다. 먼저 주어진 문장이 어떤 단어들을 포함하고 있는지 확인한다. C 언어에서 사용하는 의미 있는 단어를 '토큰token'이라고 한다. 결국 C 언어를 기계어로 번역하기 위해 첫 번째로 해야 할 일은 C 언어에서 어떤 토큰이 사용되었는지 구분하는 것인데, 컴파일러에서는 이를 어휘 분석이라 한다.

두 번째로는 이러한 단어가 문장의 구성 요소 가운데 어디에 맞는지 확인해야 한다. 즉 문법을 보면서 토큰들이 문법에 맞는지 검사하는데, 컴파일러에서는 이를 구문 분석이라 한다. 세 번째로는 한국어의 의미를 검사하는 것, 즉 의미 분석을 한다. 하지만 형식 언어는 의미를 가지고 있지 않기 때문에 이 과정에서 주로 형type을 검사한다. 의미 분석으로 생성된 코드의 실행 시간을 줄이거나 메모리를 줄이기 위한 코드 최적화를 하고, 마지막으로 목적 기계target machine의 특성을 고려하여 목적 코드를 생성해낸다.

컴파일러의 논리적 구조를 설명하는 방법은 많지만 여기서는 전단부front-end와 후단부back-end로 나누어 설명한다. 전단부는 목적 기계에 독립적인 부분으로 목적 기계에 관계없이 소스 프로그램을 분석하고 중간 코드를 생성하며, 후단부는 목적 기계에 의존적인 부분으로 전단부에서 생성한 중간 코드를 특정 기계에 대한 목적 코드로 번역한다.

컴파일러를 전단부와 후단부로 나눈 논리적 구조는 [그림 2-2]와 같다.

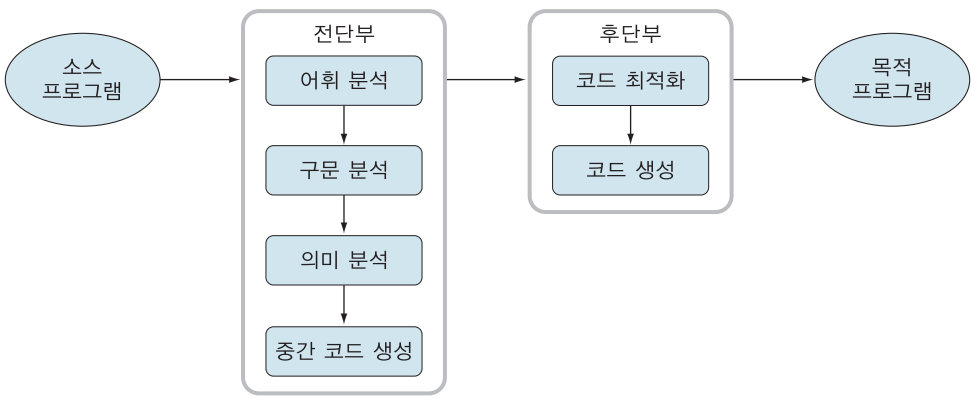


그림 2-2 **컴파일러의 구체적 구조**

컴파일러의 전단부는 문법 이론grammar theory에 의해 잘 정립된 반면, 후단부는 아직도 계속 연구가 진행 중이다.

더 세분하면 컴파일러는 소스 프로그램을 한 표현에서 다른 표현으로 변환하는 각 단계phase(컴퓨터에서 소스 프로그램의 어떤 표현을 다른 표현으로 변환하는 것)에 의해 동작한다. 컴

파일러를 단계별로 분리한 전형적인 논리적 구조는 [그림 2-3]과 같다. 몇 개의 단계를 묶어서 패스pass라고 하는데, 하나의 패스는 한 번 읽어서scanning 출력을 발생시키는 것을 의미한다. 컴파일러의 각 단계를 [그림 2-2]와 같이 전단부와 후단부로 2개의 패스로 나누면 2-패스two pass 컴파일러가 된다.

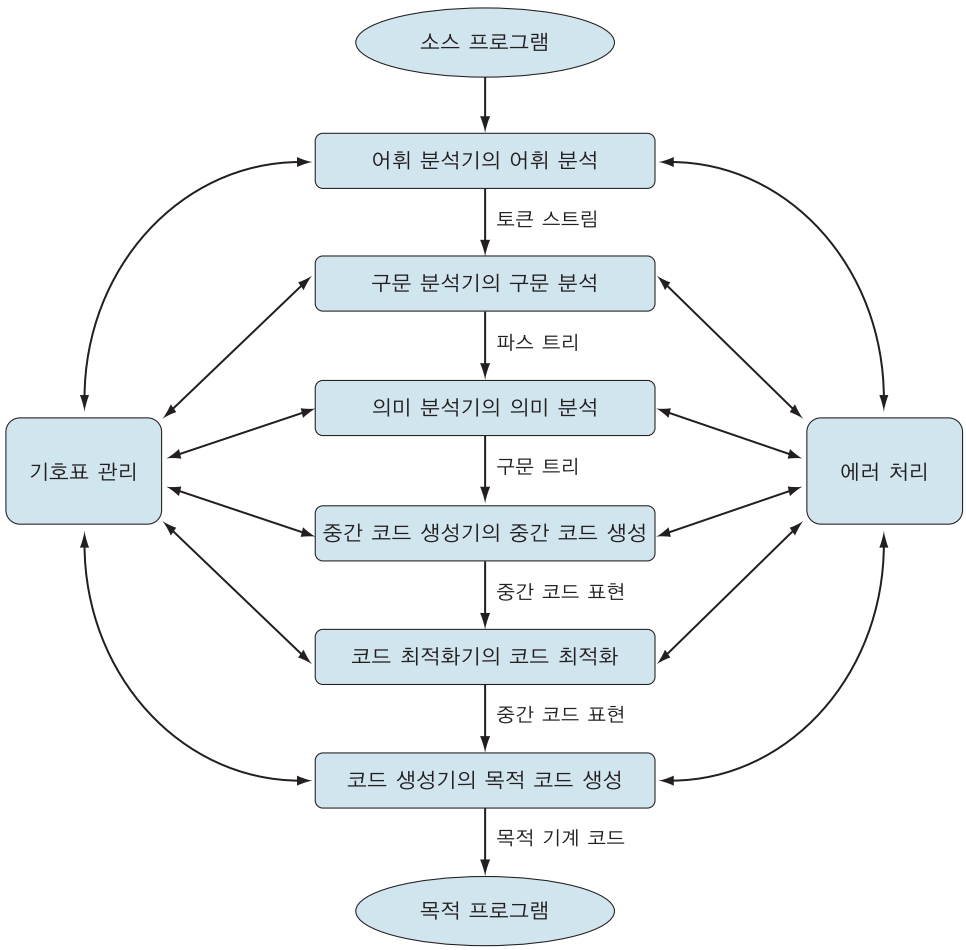


그림 2-3 **컴파일러의 논리적 구조**

2. 논리적 구조

컴파일러의 논리적 구조는 [그림 2-3]과 같이 여섯 단계로 나눌 수 있다. 각 단계에 대해서는 예제를 통해 설명할 것이다. 기호표symbol table 관리와 에러 처리error handler 기능은 여섯 단계 모두에서 상호 작용을 한다.

컴파일러에 대해 논의하려면 소스 프로그램을 작성하는 고급 언어의 문법이 필요하다. 여기서는 가장 간단한 부분적인 C 문법을 통해 살펴보자. [그림 2-4]는 C 문법을 EBNF 형태로

표현한 것으로 각각의 번호는 문법을 설명하기 위한 생성 규칙 번호이다.

TIP EBNF는 3장에서 자세히 다룰 것이다.

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**

생성 규칙 ❶은 ⟨Sub C⟩가 ⟨assign−st⟩로 치환된다는 것을 의미한다. 그리고 생성 규칙 ❷에서 ⟨assign−st⟩는 ⟨lhs⟩가 먼저 오고 =가 그다음으로 오며 ⟨exp⟩가 온 뒤 마지막으로 ;이 온다. 이와 같이 문법을 따라가보면 [그림 2−4]는 C 언어의 치환문을 표현하며, 산술식에는 산술 연산자 +, −, *, / 등 사칙 연산을 허용하는데 우선순위는 *와 /가 높고 그다음이 +, −이며, 모든 연산자가 왼쪽 결합 법칙을 취함을 알 수 있다. 또한 산술식에 괄호를 사용할 수 있으며, 식별자는 첫 글자가 영문자나 언더바로 시작하고 두 번째 글자부터는 영문자, 숫자, 언더바가 올 수 있으며 길이의 제한이 없다. 그리고 숫자는 0과 양의 정수를 사용할 수 있으며, 영문자는 소문자 a부터 z까지 사용 가능하다.

그렇다면 다음 예제에서 치환문 ni = (po − 60);이 주어진 문법에 맞는 문장인지 확인해보자. 뒤에서 설명하겠지만 이를 확인하는 방법은 크게 두 가지가 있는데 여기서는 좌단 유도 leftmost derivation 방법으로 확인해본다. 유도는 생성 규칙의 왼쪽을 오른쪽으로 대체하는 것이며, 좌단 유도는 가장 왼쪽 기호부터 유도하는 것을 말한다.

예제 2−1 문법에 맞는 문장인지 확인하기 1

치환문 ni = (po − 60);이 [그림 2−4]의 문법에 맞는 문장인지 알아보자.

▼ 풀이

시작 기호로부터 생성되는 문자열을 살펴보면 다음과 같다.

⟨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)
⇒ 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 ⓫ 적용)

즉 치환문 ni = (po − 60);은 [그림 2−4]의 문법에 맞는 문장이다.

예제 2−2 **문법에 맞는 문장인지 확인하기 2**

치환문 ni = ba * po − 60 + ni / (abc + 50);이 [그림 2−4]의 문법에 맞는 문장인지 알아보자.

▼ 풀이

시작 기호로부터 생성되는 문자열을 살펴보면 다음과 같다.

⟨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 ⑩ 적용)
⇒ ni = 〈exp〉+〈term〉; (∵ 생성 규칙 ④ 적용)
⇒ ni = 〈exp〉−〈term〉+〈term〉; (∵ 생성 규칙 ④ 적용)
⇒ ni = 〈term〉−〈term〉+〈term〉; (∵ 생성 규칙 ④ 적용)
⇒ ni = 〈term〉∗〈factor〉−〈term〉+〈term〉; (∵ 생성 규칙 ⑤ 적용)
⇒ ni = 〈factor〉∗〈factor〉−〈term〉+〈term〉; (∵ 생성 규칙 ⑤ 적용)
⇒ ni = 〈variable〉∗〈factor〉−〈term〉+〈term〉; (∵ 생성 규칙 ⑥ 적용)
⇒ ni = 〈ident〉∗〈factor〉−〈term〉+〈term〉; (∵ 생성 규칙 ⑦ 적용)
⇒ ni = (〈letter〉|_){〈letter〉|〈digit〉|_}∗〈factor〉−〈term〉+〈term〉;
(∵ 생성 규칙 ⑧ 적용)
⇒ ni = b{〈letter〉|〈digit〉|_}∗〈factor〉−〈term〉+〈term〉;
(∵ 생성 규칙 ⑩ 적용)
⇒ ni = ba ∗ 〈factor〉−〈term〉+〈term〉; (∵ 생성 규칙 ⑩ 적용)
⇒ ni = ba ∗ 〈variable〉−〈term〉+〈term〉; (∵ 생성 규칙 ⑥ 적용)
⇒ ni = ba ∗ 〈ident〉−〈term〉+〈term〉; (∵ 생성 규칙 ⑦ 적용)
⇒ ni = ba ∗ (〈letter〉|_){〈letter〉|〈digit〉|_}−〈term〉+〈term〉;
(∵ 생성 규칙 ⑧ 적용)
⇒ ni = ba ∗ p{〈letter〉|〈digit〉|_}−〈term〉+〈term〉;
(∵ 생성 규칙 ⑩ 적용)
⇒ ni = ba ∗ po − 〈term〉+〈term〉; (∵ 생성 규칙 ⑩ 적용)
⇒ ni = ba ∗ po − 〈factor〉+〈term〉; (∵ 생성 규칙 ⑤ 적용)
⇒ ni = ba ∗ po − 〈number〉+〈term〉; (∵ 생성 규칙 ⑥ 적용)
⇒ ni = ba ∗ po − {〈digit〉}+〈term〉; (∵ 생성 규칙 ⑨ 적용)
⇒ ni = ba ∗ po − 60 + 〈term〉; (∵ 생성 규칙 ⑪ 적용)
⇒ ni = ba ∗ po − 60 + 〈term〉/〈factor〉; (∵ 생성 규칙 ⑤ 적용)
⇒ ni = ba ∗ po − 60 + 〈factor〉/〈factor〉; (∵ 생성 규칙 ⑤ 적용)
⇒ ni = ba ∗ po − 60 + 〈variable〉/〈factor〉; (∵ 생성 규칙 ⑥ 적용)
⇒ ni = ba ∗ po − 60 + 〈ident〉/〈factor〉; (∵ 생성 규칙 ⑦ 적용)
⇒ ni = ba ∗ po − 60 + (〈letter〉|_){〈letter〉|〈digit〉|_}/〈factor〉;
(∵ 생성 규칙 ⑧ 적용)
⇒ ni = ba ∗ po − 60 + n{〈letter〉|〈digit〉|_}/〈factor〉;
(∵ 생성 규칙 ⑩ 적용)
⇒ ni = ba ∗ po − 60 + ni /〈factor〉; (∵ 생성 규칙 ⑩ 적용)
⇒ ni = ba ∗ po − 60 + ni /〈exp〉; (∵ 생성 규칙 ⑥ 적용)
⇒ ni = ba ∗ po − 60 + ni /(〈exp〉+〈term〉); (∵ 생성 규칙 ④ 적용)
⇒ ni = ba ∗ po − 60 + ni /(〈term〉+〈term〉); (∵ 생성 규칙 ④ 적용)
⇒ ni = ba ∗ po − 60 + ni /(〈factor〉+〈term〉); (∵ 생성 규칙 ⑤ 적용)
⇒ ni = ba ∗ po − 60 + ni /(〈variable〉+〈term〉); (∵ 생성 규칙 ⑥ 적용)
⇒ ni = ba ∗ po − 60 + ni /(〈ident〉+〈term〉); (∵ 생성 규칙 ⑦ 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)

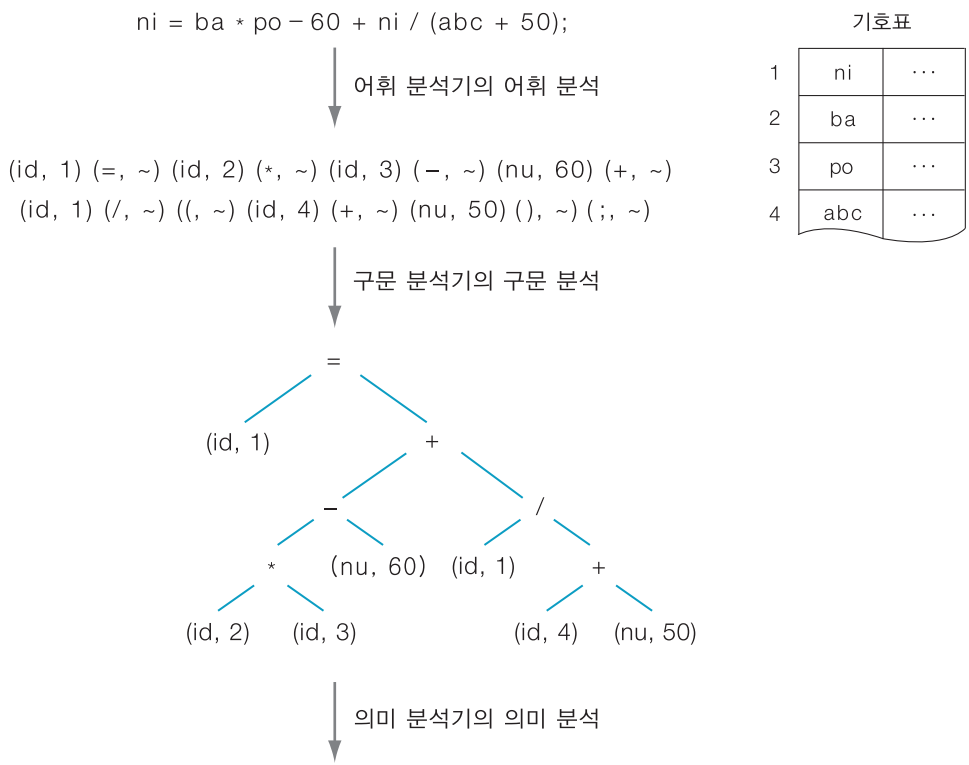
⇒ ni = ba * po − 60 + ni /(abc+⟨number⟩); (∵ 생성 규칙 ❻ 적용)

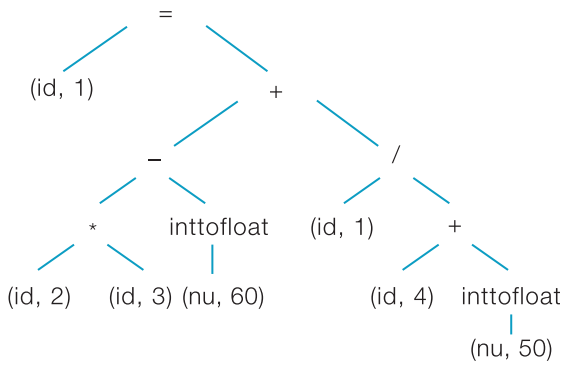
⇒ ni = ba * po − 60 + ni /(abc+{⟨digit⟩}); (∵ 생성 규칙 ❾ 적용)

⇒ ni = ba * po − 60 + ni /(abc+50); (∵ 생성 규칙 ⓫ 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.

[예제 2-2]의 치환문에 대한 컴파일러 논리적 구조의 각 단계별 전체 내용은 [그림 2-5]와 같다. 이는 치환문 ni = ba * po − 60 + ni / (abc + 50);에 대한 컴파일러의 전체 과정을 도식화한 것이다.





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

그림 2-5 **치환문의 컴파일링 과정**

2.1 어휘 분석(스캐닝)

컴파일러의 첫 번째 단계를 어휘 분석lexical analysis 또는 스캐닝scanning이라 한다. 어휘 분석이란 소스 프로그램을 읽어들여 토큰이라는 의미 있는 문법적 단위로 분리하고 토큰 스트림token stream을 생성하는 것이다. 이러한 어휘 분석을 담당하는 도구를 어휘 분석기lexical analyzer 또는 스캐너scanner라고 한다.

언어마다 사용하는 토큰이 다르지만 일반적인 프로그래밍 언어에서 사용하는 토큰은 if, for, while과 같은 예약어reserved word, 3, 2.5와 같은 상수constant, +, −, *, /, =와 같은 연산자operator, 프로그래머가 정의한 식별자identifier, 그리고 괄호나 쉼표(,), 세미클론(;)과 같은 구분자delimiter 등이 있다.

이러한 토큰은 다음 단계인 구문 분석에서 효율을 높이기 위해 순서쌍 (토큰 번호, 속성 값)의 형태로 전달한다. 토큰 번호token number는 모든 토큰을 구별하기 위한 유일한 번호이고, 두 번째 요소인 속성 값attribute value은 기호표에 저장된 항목을 가리킨다.

토큰을 구별하려면 주어진 문법에 어떤 토큰을 사용하고 있는지를 알아야 한다. 일반적인 프로그래밍 언어에서는 앞서 설명한 바와 같이 다섯 가지 토큰, 즉 예약어, 상수, 연산자, 식별자, 구분자 등이 사용된다.

[그림 2-4]의 문법을 살펴보면 예약어를 사용하지 않는다. 상수는 0이나 양의 정수를 허용하며, 연산자는 산술 연산자인 +, −, *, /를 사용하고 치환 연산자인 =를 허용한다. 식별자도 허용하며, 첫 자는 영문자나 언더바로 시작하고 둘째 자부터는 영문자, 숫자, 언더바를 사용하며 길이에 제한이 없다. 구분자는 세미콜론과 여는 괄호, 닫는 괄호를 사용하고 있다. 이를 정규 문법과 정규 표현으로 나타내면 다음과 같다.

① **식별자** : 정규 문법

$S \rightarrow lA \mid \_A$

$A \rightarrow lA \mid dA \mid _A \mid \varepsilon$으로부터 정규 표현으로 나타내기 위한 방정식은 다음과 같다.

$S = (l + \_)A$

$A = lA \mid dA \mid \_A \mid \varepsilon = (l + d + \_)A + \varepsilon = (l + d + \_)^*$

이것으로부터 해를 구하면

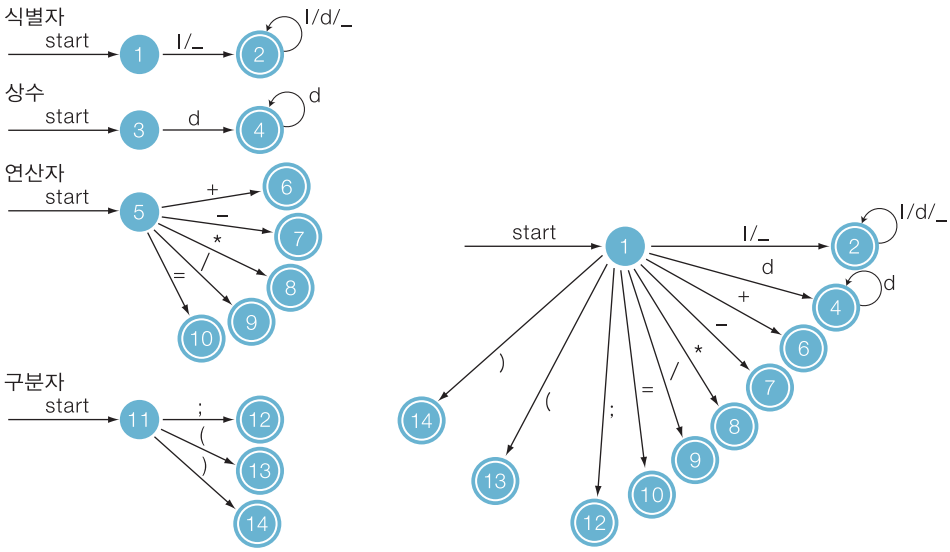
$S = (l + \_)A = (l + \_)(l + d + \_)^*$

② **상수** : $S = (d)^+$

③ **연산자** : $S = + \mid - \mid * \mid / \mid =$

④ **구분자** : ;, (,)

이것들을 받아들이는 유한 오토마타는 [그림 2-6]과 같다.



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

그림 2-6 **[그림 2-4]에 대한 어휘 분석기**

[그림 2-6]과 같이 어휘 분석기를 구성하는 방법은 문법을 통해 다섯 가지 토큰 중 어느 것이 사용되고 있는지를 확인한 다음, 사용되는 토큰을 정규 표현으로 나타내는 것이다. 그리고 그 정규 표현을 받아들이는 유한 오토마타를 만들면 된다.

[그림 2-6]과 같은 어휘 분석기가 구현되면 소스 프로그램을 읽으면서 치환문 ni = ba * po – 60 + ni / (abc + 50);은 다음의 16개 토큰 스트림으로 분리될 것이다. 이때 토큰은 토큰 번호가 출력될 것이다. 예를 들어 =이면 10이 출력되는데 여기서는 설명을 위해 그냥 =로 출력한다.

① 식별자 ni는 토큰 (id, 1)로 출력된다. 여기서 id는 식별자이고 1은 ni의 기호표 항목을 가리킨다. 식별자에 대한 기호표는 이름과 형 같은 식별자에 대한 정보를 보유한다.

② 치환 연산자 =는 토큰 (=, ~)로 출력한다. 여기서 ~는 null 값이다.

③ 식별자 ba는 토큰 (id, 2)로 출력한다. 여기서 2는 기호표 항목이다.

④ 곱하기 연산자 *는 토큰 (*, ~)로 출력한다.

⑤ 식별자 po는 토큰 (id, 3)으로 출력한다. 여기서 3은 기호표 항목이다.

⑥ 빼기 연산자 –는 토큰 (–, ~)로 출력한다.

⑦ 상수 60은 토큰 (nu, 60)으로 출력한다.

⑧ 더하기 연산자 +는 토큰 (+, ~)로 출력한다.

⑨ 식별자 ni는 (id, 1)로 출력한다.

⑩ 나누기 연산자 /는 토큰 (/, ~)로 출력한다.

⑪ 구분자 (는 토큰 ((, ~)로 출력한다.

⑫ 식별자 abc는 (id, 4)로 출력한다.

⑬ 더하기 연산자 +는 토큰 (+, ~)로 출력한다.

⑭ 상수 50은 토큰 (nu, 50)으로 출력한다.

⑮ 구분자)는 토큰 ( ), ~)로 출력한다.

⑯ 구분자 ;은 토큰 (;, ~)로 출력한다.

토큰을 분리하는 공백blank은 어휘 분석기에 의해 무시되며, 어휘 분석이 끝나면 다음과 같은 토큰 스트림을 보여준다.

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

이를 토큰 번호와 토큰 값으로 나타내면 다음과 같다.

(2, 1) (10, ~) (2, 2) (8, ~) (2, 3) (7, ~) (4, 60) (6, ~) (2, 1) (9, ~) (13, ~) (2, 4) (6, ~) (4, 50) (14, ~) (12, ~)

어휘 분석에 관한 자세한 내용은 4장에서 다룰 것이다.

2.2 구문 분석(파싱)

컴파일러의 두 번째 단계는 구문 분석syntax analysis 또는 파싱parsing이라 한다. 어휘 분석 단계의 결과인 토큰 스트림을 받아서 주어진 문법에 맞는지 검사하며, 주어진 문법에 맞는 문장의 경우 그 문장에 대한 파스 트리parse tree를 출력하고 올바르지 않은 문장의 경우 에러 메시지를 출력한다. 이처럼 구문 분석을 담당하는 도구를 구문 분석기syntax analyzer 또는 파서parser라고 부른다.

일반적으로 파스 트리는 토큰을 터미널 노드terminal node로 하는 트리이다. [그림 2-7]은 치환문 ni = ba * po − 60 + ni / (abc + 50);에 대한 파스 트리이다. 치환문 ni = ba * po − 60 + ni / (abc + 50);은 이미 어휘 분석에서 치환문 id = id * id − nu + id / (id + nu);로 생성되었다.

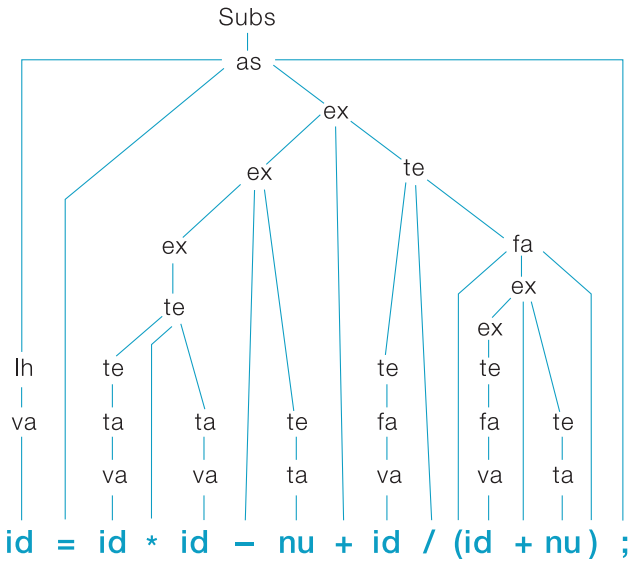


그림 2-7 치환문 ni = ba * po − 60 + ni / (abc + 50);의 파스 트리

치환문 ni = ba * po − 60 + ni / (abc + 50);에서 [그림 2-4]의 문법을 보면 치환문의 우선순위와 결합 법칙이 적용되어 있다. 연산자의 우선순위는 *와 /가 +와 −보다 높고 4개의 연산자 모두 왼쪽 결합 법칙을 가지고 있다. 괄호를 이용하여 연산 순서를 표현하면 다음과 같다.

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

즉 ba * po를 먼저 계산하고 그다음에 ((ba * po) − 60), (abc + 50), (ni / (abc + 50)), (((ba * po) − 60) + (ni / (abc + 50)))을 계산한다.

이와 같은 파스 트리는 중간 코드 생성 단계에서 입력으로 이용된다. 일반적으로 파스 트리는 [그림 2-7]에서 보는 바와 같이 다음 단계에서 사용하지 않는 불필요한 정보인 논터미널 기호 variable, lhs, factor, term, exp, assign-st 등을 가지고 있으므로 기억 공간을 낭비하고 컴파일 시간을 증가시키는 요인이 된다. 따라서 이러한 불필요한 정보를 제거하고 다음 단계에서 꼭 필요한 정보만으로 구성된 트리를 만드는데, 이를 추상 구문 트리abstract syntax tree 혹은 간단히 구문 트리syntax tree라고 한다. 이 책에서는 구문 트리라는 용어를 사용한다.

트리는 연산자를 내부 노드에 나타내고, 그 연산자에 대한 피연산자를 연산자의 자식 노드children node에 나타내는 구조를 가진 파스 트리의 축약된 형태이다. [그림 2-7]의 파스 트리를 구문 트리로 만들면 [그림 2-8]과 같다. 여기서 구분자 세미콜론과 괄호는 아무 의미가 없으므로 제거된다.

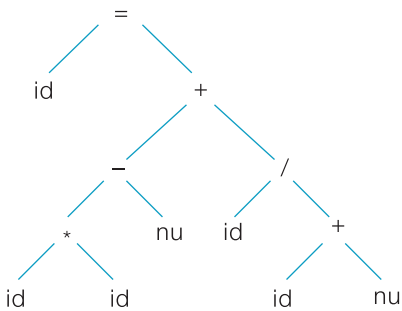


그림 2-8 [그림 2-7]의 파스 트리에 대한 구문 트리

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

표 2-1 [그림 2-4]의 문법에 대한 SLR 파싱표

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
10			r13	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							

[표 2-1]의 파싱표를 가지고 치환문 ni = (((ba * po) − 60) + (ni / (abc + 50)));에 대해 파싱하는 과정은 [표 2-2]와 같다.

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
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

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
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.

구문 분석은 6장에서 자세히 설명할 것이다.

2.3 의미 분석

의미 분석semantic analysis에서는 구문 트리와 기호표에 있는 정보를 이용하여 소스 프로그램이 언어 정의에 의미적으로 일치하는지를 검사하고, 다음 단계인 중간 코드 생성에 이용하기 위해 자료형 정보를 수집하여 구문 트리나 기호표에 저장한다. 이와 같은 일을 담당하는 도구를 의미 분석기semantic analyzer라고 한다.

그런데 자연 언어에서는 의미 분석이 의미가 있지만 우리가 다루는 형식 언어는 의미를 가지고 있지 않으므로 의미 분석이 크게 중요하지 않다. 따라서 의미 분석에서 가장 중요한 일 중의 하나는 형 검사type checking이다. 형 검사란 각 연산자가 소스 프로그램 규칙에 의해 허용된 피연산자를 가졌는지 검사하는 것이다. 예를 들면 어떤 프로그래밍 언어에서 배열의 첨자는 정수형만을 허용하는데 배열의 첨자에 부동 소수형을 사용했다면 의미 분석기가 에러임을 지적한다.

일반적으로 수식에 대한 연산은 크게 두 가지 종류가 있다. 하나는 프로그래밍 언어에서 피연산자와 연산 결과의 형이 고정되어 있는 연산, 즉 형 고정 연산type specific operation이고, 또 하나는 프로그래밍 언어에서 피연산자와 연산 결과의 형이 변할 수 있는 연산, 즉 일반적 연산generic operation이다.

예를 들어 a + b와 같은 산술식을 생각해보자. a와 b의 자료형이 각각 int형과 float형이라면 형 고정 연산에서는 에러 메시지를 주지만, 일반적 연산에서는 a나 b의 형을 변환하여 연산을 허용한다. 형 변환type conversion은 주어진 자료형의 값을 다른 자료형의 값으로 변환하는 것을 말하며, 시스템에서 자동으로 형을 변환하는 묵시적 형 변환implicit type conversion과 프로그래머가 명시적으로 형 변환을 요구하는 명시적 형 변환explicit type conversion이 있다. 명시적 형 변환은 프로그래머가 명령문으로 요구한 형 변환으로 캐스트cast 연산이라고 일컫는다. 그리고 묵시적 형 변환은 시스템에서 자동으로 형을 변환하는 것으로 자동 변환automatic type conversion 또는 강제 형 변환coercion이라고 한다.

어떤 언어에서 일반적 연산을 허용한다고 하자. 예를 들어 산술식 a+b에서 a는 int형, b는 float형이라면 시스템은 자동 변환을 할 것이다. 자동 변환을 하려면 형을 변환해야 하는데, 이는 컴파일러가 판단해서 결정하는 것이 아니라 언어 정의language definition 시간에 이미 어떤 자료형으로 할 것인지 결정해 놓는다. 대부분의 언어는 이때 정수형인 int를 실수형인 float로 형 변환을 하며, 이런 자동 변환을 의미 분석 단계에서 처리한다. 더 자세한 내용은 7장에서 다룰 것이다.

예제 2-3 문장의 의미 분석

어떤 언어가 일반적 연산을 허용하고 정수형인 int와 실수형인 float가 있는 경우, 언어 정의 시간에 정수형을 실수형으로 변환하여 계산한다고 가정하고, [그림 2-6]의 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 분석을 해보자.

▼ 풀이

의미 분석을 하면 [그림 2-9]와 같다. 여기서 inttofloat는 정수형을 실수형으로 변환하는 함수이다. inttofloat의 피연산자가 상수이기 때문에 컴파일러는 정수형 상수 대신 동일한 실수형 상수 값으로 대치할 수 있다.

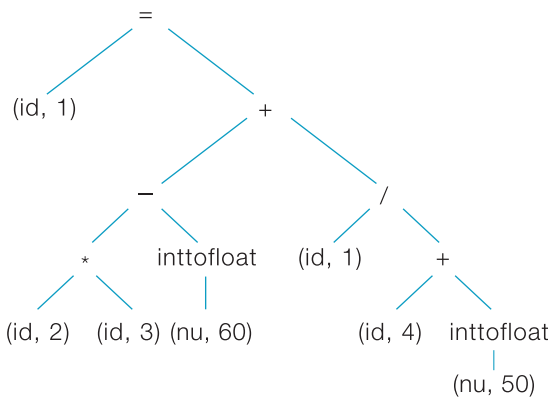


그림 2-9 **[그림 2-8]에 대한 의미 분석 후의 구문 트리**

의미 분석과 형 검사는 7장에서 자세히 설명할 것이다.

2.4 중간 코드 생성

중간 코드 생성intermediate code generation 단계에서는 구문 트리를 이용하여 각 구문에 해당하는 중간 코드를 생성하거나 한 문법 규칙이 감축될 때마다 구문 지시적 번역syntax-directed translation이 이뤄진다. 구문 지시적 번역은 문법 규칙이 감축될 때 그 규칙에 알맞은 코드 생성 루틴을 호출함으로써 중간 코드를 생성하는 것이다. 이러한 중간 코드 생성을 담당하는 도구를 중간 코드 생성기intermediate code generator라고 부른다.

치환문 ni = ba * po − 60 + ni / (abc + 50);에 대해 중간 언어 중에 3-주소 코드three-address code를 이용한다면 다음과 같은 중간 코드가 생성된다. 3-주소 코드는 각 명령어마다 3개의 피연산자를 가진 명령어의 나열이다.

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

여기서 컴파일러는 각 명령어의 수행에 의해 계산된 값을 담고 있을 임시 메모리인 temp1, temp2, temp3, temp4, temp5, temp6, temp7과 같은 임시 변수를 생성해야 한다.

3-주소 코드 명령어에 대해 주목할 사항이 있다. 첫째, 각 3-주소 코드는 우변에 기껏해야 하나의 연산자를 갖는다. 그러므로 이들 명령어는 연산이 수행되는 순서를 고정시킨다. 둘째, 컴파일러는 3-주소 명령어에 의해 계산된 값을 저장하기 위해 임시 변수를 생성한다.

8장에서 중간 언어와 중간 코드 생성에 대해 자세히 설명할 것이다.

2.5 코드 최적화

코드 최적화code optimization는 주어진 입력 프로그램과 의미적으로 동등하면서 좀 더 효율적인 코드로 바꾸어 코드 실행 시 기억 공간이나 실행 시간을 절약하기 위한 단계이다. 이는 선택적인 단계로서 생략되는 경우도 있지만, 최근에는 컴퓨터 구조를 간단히 하고 기존에 컴퓨터 구조가 담당하던 많은 일을 컴파일러에 넘김으로써 RISCreduced instruction set computer와 같은 컴퓨터 시스템에서는 컴퓨터 특성을 활용하기 위해 최적화 단계를 많이 사용하고 있다.

코드 최적화는 개선 대상에 따라 여러 가지로 분류할 수 있다. 최적화가 적용되는 프로그램의 영역에 따라 지역 최적화local optimization, 전역 최적화global optimization, intra-procedural optimization, 프로시저 간 최적화inter-procedural optimization로 나눌 수 있으며, 기능적인 측면에서는 실행 시간 최적화와 메모리 최적화로 나눌 수 있다. 또한 최적화가 많이 이뤄지는 부분에 따라 루프(혹은 반복문) 최적화와 단일문 최적화로 구분되고, 목적 기계의 의존성에 따라서는 기계 독립 최적화machine independent optimization와 기계 종속 최적화machine dependent optimization 등이 있다.

지역 최적화는 부분적인 관점에서 일련의 비효율적인 코드를 구분해내고 좀 더 효율적인 코드로 만드는 방법으로, 코드가 분기해 나가거나 분기해 들어오는 부분이 없는 기본 블록basic block 내에서 최적화가 이뤄지므로 상대적으로 쉽게 수행할 수 있다. 기본 블록은 어떠한 제어 흐름도 가지지 않기 때문에 분석이 거의 필요 없다. 이 방법에는 공통 부분식 제거common subexpression elimination, 복사 전파copy propagation, 죽은 코드 제거dead-code elimination, 상수 폴딩, 대수학적 간소화 등이 있다.

전역 최적화는 기본 블록을 넘어서지만 하나의 프로시저 내에서 일련의 비효율적인 코드를 구분해내고 좀 더 효율적인 코드로 만드는 방법이다. 이 방법을 적용하는 데는 지역 최적화보다 더 많은 정보와 비용이 필요하여 수행하기가 어렵지만 지역 최적화보다 효과가 훨씬 좋다. 전역 최적화는 일반적으로 자료 흐름 분석data flow analysis이라는 기법을 사용하는데, 이는 프로그램 안에서 사용되는 변수의 통로path에 관한 정보를 수집하여 처리하는 과정이다. 이러한 방법에는 전역적 공통 부분식 제거, 상수 폴딩, 도달 불가능한 코드 제거 등이 있다.

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**

10장에서 코드 최적화에 대해 자세히 설명할 것이다.

2.6 목적 코드 생성

목적 코드 생성object code generation은 컴파일 과정의 마지막 단계로, 연산을 수행할 레지스터를 선택하거나 자료에 메모리의 위치를 정해주며 실제로 목적 기계에 맞는 코드를 생성한다.

[그림 2−10]에 대해 레지스터 2개(R1, R2)를 사용하여 코드를 변환하면 다음과 같다.

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

각 명령어의 첫 번째 피연산자는 목적지destination를 나타내며 #60.0과 #50.0은 상수로 취급되고 있음을 보여준다. 한편 목적 코드 생성에서 소스 프로그램에 있는 식별자의 메모리 할당과 레지스터의 운영에 대해서는 무시했다. 12장에서 목적 코드 생성에 대해 자세히 살펴볼 것이다.

지금까지 살펴본 바와 같이 컴파일 과정은 논리적으로 6단계로 이뤄지며 이와 관련된 기호표 관리와 에러 처리가 있다.

컴파일러는 소스 프로그램에 나타난 모든 자료에 대한 정보를 가지고 있어야 한다. 예를 들어 어떤 변수가 정수를 나타내는지 실수를 나타내는지, 배열의 크기가 얼마나 되는지, 함수의 매개변수가 몇 개 필요한지를 알아야 한다. 이와 같은 자료에 대한 정보는 어휘 분석, 구

문 분석 단계에서 수집되어 기호표에 저장된다. 기호표와 관련된 내용은 7장에서 자세히 설명할 것이다.

또한 컴파일러는 소스 프로그램에서 에러를 발견하면 사용자에게 알려주어야 한다. 예를 들어 어휘 분석 단계에서 소스 프로그램에서 사용할 수 없는 문자를 사용한다든가, 구문 분석 단계에서 괄호가 빠지는 것과 같은 문법 규칙의 에러가 발생할 수 있다. 컴파일러는 이와 같은 에러를 처리해주는 에러 처리 루틴error handling routine을 가지고 있다.

02 컴파일러의 물리적 구조

컴파일러의 논리적 구조라 함은 이론적으로 컴파일러를 어떤 요소로 구성해야 한다는 의미이지만 실제의 컴파일러가 꼭 이렇게 구성되는 것은 아니다. 예를 들면 어휘 분석과 구문 분석에 대해 어떤 컴파일러에서는 먼저 어휘 분석 루틴이 작동하여 소스 프로그램 전부를 토큰의 열로 변환한 뒤 구문 분석 루틴을 호출하지만, 또 다른 컴파일러에서는 먼저 구문 분석 루틴이 작동하여 어휘 분석 루틴을 부프로그램으로 호출하기도 한다. 이처럼 컴파일러의 논리적 구조를 실제로 구현하는 경우인 물리적 구조physical organization가 반드시 논리적 구조의 순서와 일치하지는 않는다.

컴파일러의 물리적 구조에서 가장 중요한 사항은 컴파일러를 어떻게 구현할 것인지, 그리고 컴파일러를 구현하기 위해 설계할 때 고려해야 할 사항이다. 그렇다면 먼저 컴파일러를 어떻게 구현할 것인지를 살펴보자.

컴파일러를 어떻게 구현하느냐에서는 일반적으로 여러 단계를 모아 하나의 패스를 만든다. 패스는 하나의 입력 파일을 단 한 번만 읽는 것으로 구성되는 단위로, 컴파일러는 대부분 1-패스 컴파일러나 2-패스 컴파일러로 구현한다. 1-패스 컴파일러는 초창기에 컴파일러를 만들 때 사용했던 방법으로 컴파일러의 논리적 단계를 하나의 패스로 구현하는 것이고, 2-패스 컴파일러는 중간 코드를 기점으로 앞부분을 전단부, 뒷부분을 후단부로 나누어 구성하는 방법이다. 따라서 컴파일러는 이러한 패스에 의해 구현되며, [그림 2-11]은 패스로 구성된 컴파일러의 물리적 구조를 보여준다.

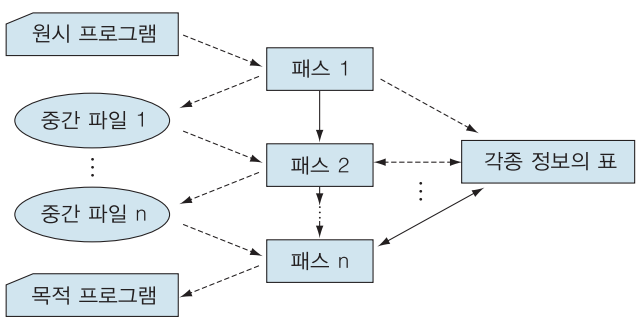


그림 2-11 **컴파일러의 물리적 구조**

1-패스 컴파일러는 초창기에 컴파일러 이론이 정립되지 않은 상태에서 설계된 컴파일러로 소스 프로그램을 중간 코드 생산 없이 직접 목적 코드로 번역했으며, 그 후 컴파일러 설계에

대한 이론이 정립되어감에 따라 점차 번역 단계가 세분화되어 컴파일러를 여러 모듈로 나누어 설계할 수 있게 되었다. 이에 따라 각 모듈을 연결해주는 중간 코드가 필요하게 되었으며, 최근에는 중간 코드 선정 문제가 컴파일러 설계에서 큰 몫을 담당하고 있다. 이 가운데 2-패스 컴파일러는 어휘 분석, 구문 분석, 의미 분석, 중간 코드 생성을 전단부로, 코드 최적화와 목적 코드 생성을 후단부로 나누는 방법이다.

1-패스 컴파일러는 전진 분기forward jump의 처리를 위해 빈칸으로 남겨져 있다가 그 정보를 얻었을 때 채워넣는 백패칭back patching이 필요하나, 컴파일러 설계 시부터 목적 기계를 고려해서 만들 수 있기 때문에 효율성이 좋고 실행 속도가 빠르다는 장점이 있다. 반면에 2-패스 컴파일러는 기능적으로 독립된 여러 모듈로 컴파일러를 구성할 수 있기 때문에 이식성이 좋고, 중간 코드를 이용한 최적화에 따라서 기계와 독립적인 최적화가 가능하며, 하나의 패스가 사용했던 공간을 다시 사용할 수 있으므로 메모리를 절약할 수 있다. 그러나 직접 코드로 번역하지 못하기 때문에 기계 코드의 표현에 제약이 있고 실행 속도가 느리다는 것이 단점이다. 그렇다 해도 컴파일러를 만드는 일이 쉽지 않고 많은 언어와 목적 기계가 존재하므로 이식성을 고려하여 일반적으로 2-패스 컴파일러를 많이 사용하고 있다.

컴파일러를 구현하는 것은 일반 소프트웨어를 구현하는 것과 비슷하다. 다음은 컴파일러를 구현할 때 고려해야 할 사항이다.

■ 설계 문제

각각의 패스에 어떤 기능을 할당하고 패스를 모두 몇 개로 할 것인가 등 설계상의 문제이다. 이를 결정하는 근거 및 조건이 되는 것은 컴파일러의 논리적 구조, 컴퓨터 자원, 사용자의 요구 사항, 컴파일러를 개발하는 인적 자원 등이다. 컴파일러의 논리적 구조는 필연성에서 얻어진 것이므로 되도록 물리적 구조를 거기에 맞추는 것이 좋다. 즉 어휘 분석, 구문 분석, 의미 분석, 중간 코드 생성, 코드 최적화, 코드 생성 등의 단계를 하나로 총괄해서 1-패스로 만들거나, 어휘 분석부터 중간 코드 생성까지를 하나의 패스로 묶어서 2-패스로 만들거나, 아니면 더 세부적으로 여러 개의 패스로 만들 수 있다. 이는 구현하는 사람이 주변 조건 등을 고려하여 만들겠지만 가능한 한 컴파일러의 여섯 가지 단계를 갖춰야 한다.

■ 컴퓨터 자원 문제

컴파일러에 가장 큰 영향을 주는 것은 메모리 용량이다. 하나의 패스가 실행되고 있을 때는 그 패스의 루틴과 그 패스에서 참조하는 정보의 표가 모두 주기억장치에 있지 않으면 효율이 떨어진다. 따라서 주기억장치의 용량이 작으면 패스의 수를 늘려서 각 패스의 크기를 작게 하지 않으면 안 된다. 즉 1-패스 컴파일러보다는 2-패스 컴파일러로, 2-패스 컴파일러보다는 3-패스 컴파일러로 만들게 되는 것이다.

그러나 최근의 컴퓨터는 대부분 가상 메모리virtual memory 방식을 사용하므로 실제 용량이 컴파일러의 설계에 큰 제약이 되지 않는다고 볼 수 있다. 그렇다고 해도 가상 메모리는 실제 메모리보다 실행 효율이 떨어지므로 패스를 너무 크게 하면 안 된다. 또한 작은 마이크로컴퓨터의 경우에는 디스크와 같은 보조 메모리가 없는 것이 있는데, 이런 경우에는 하나의 패스만으로 구성해야 한다. 그 밖에 운영체제나 연결 편집기linkage editor의 기능, 컴파일러의 메타 언어meta language 등 소프트웨어 시스템도 고려해야 할 중요한 자원이다.

■ 사용자의 요구 사항

사용자가 실행 시간을 중요시하는 경우도 있고, 컴파일하는 시간을 중요시하는 경우도 있으며, 에러가 발생했을 때 디버깅하는 기능을 중요시하는 경우도 있다. 또한 컴파일러의 크기를 작게 하거나 컴파일하는 동안 메모리의 용량 등을 고려할 수도 있다. 물론 이러한 모든 경우를 만족시키는 컴파일러는 없다. 한 예로 컴파일하는 시간을 중요시하는 경우를 생각해보자. 이 경우에는 컴파일러를 설계할 때 큰 메모리 용량을 사용하여 한 번에 컴파일할 수 있도록 해야 한다. 반면에 에러가 발생했을 때의 디버깅 기능을 중요시하는 경우에는 컴파일하는 시간보다는 컴파일하는 과정에서 생겨나는 모든 정보를 간직하게 하여 나중에 에러가 발생했을 때 디버깅을 쉽게 할 수 있도록 설계해야 한다.

■ 컴파일러를 개발하는 인적 자원

일반적으로 컴파일러를 만드는 데는 많은 인력과 시간이 소요된다. 그러므로 컴파일러의 개발에 얼마만큼의 인원과 시간을 쓸 수 있는가에 따라 설계도 큰 영향을 받는다. 소프트웨어 개발에 필요한 시간과 완성된 것의 질은 개발에 참여하는 인력의 자질이나 경험에 의해 크게 좌우된다. 능력이 충분하지 않으면 모험을 피해야 한다. 또한 많은 인원을 동원할 경우에는 각자가 제멋대로 생각하지 않도록 설계상의 통일을 기하는 것이 중요하다.

기본 설계는 중심적인 사람이 하는 것이 좋다. 즉 각 패스의 기능과 레지스터 간에 주고받는 중간 언어 및 기호표의 정보를 혼자 설계한 후 각 패스의 내부 설계를 다음 단계의 사람이 분담하는 것이 바람직하다.

요약

01 컴파일러의 논리적 구조

① 컴파일러의 논리적 구조는 전단부와 후단부로 나눌 수 있다.

- 전단부 : 목적 기계에 독립적이며 소스 프로그램에 관계되는 부분으로, 소스 프로그램을 분석하고 중간 코드를 생성하는 부분
- 후단부 : 목적 기계에 의존적이며, 전단부에서 생성한 중간 코드를 특정 기계에 대한 목적 코드로 번역하는 부분

② 어휘 분석(스캐닝) : 소스 프로그램을 읽어들여 토큰이라는 의미 있는 문법적 단위로 분리하여 토큰 스트림을 생성한다.

③ 구문 분석(파싱) : 토큰 스트림을 받아서 주어진 문법에 맞는지 검사하며, 주어진 문법에 맞는 문장의 경우 그 문장에 대한 파스 트리를 출력하고 올바르지 않은 문장의 경우 에러 메시지를 출력한다.

④ 의미 분석 : 구문 트리와 기호표에 있는 정보를 이용하여 소스 프로그램이 언어 정의에 의미적으로 일치하는지를 검사하고, 다음 단계인 중간 코드 생성에 이용하기 위해 자료형 정보를 수집하여 구문 트리나 기호표에 저장한다.

⑤ 중간 코드 생성 : 구문 트리를 이용하여 각 구문에 해당하는 중간 코드를 생성하거나 한 문법 규칙이 감축될 때마다 구문 지시적 번역이 이뤄진다.

⑥ 코드 최적화 : 주어진 입력 프로그램과 의미적으로 동등하면서 좀 더 효율적인 코드로 바꾸어 코드 실행 시 기억 공간이나 실행 시간을 절약한다.

⑦ 목적 코드 생성 : 연산을 수행할 레지스터를 선택하거나 자료에 메모리의 위치를 정해주며 실제로 목적 기계에 맞는 코드를 생성한다.

02 컴파일러의 물리적 구조

컴파일러의 물리적 구조는 구현에 관련된 사항으로 설계 문제, 컴퓨터 자원 문제, 사용자의 요구 사항, 컴파일러를 개발하는 인적 자원 등을 고려해야 한다.

연습문제

01 다음 용어에 대해 설명하시오.

어휘 분석	구문 분석	의미 분석
토큰	파스 트리	코드 최적화
중간 코드 생성	목적 코드 생성	

02 치환문 ni = (po − 60);을 가지고 [그림 2-4]의 문법에 맞는 문장인지 검사하시오. 그리고 컴파일러의 논리적 구조에 대해 각 단계별로 도해하고, 각각의 단계를 [그림 2-5]와 같이 설명하시오.

03 1-패스 컴파일러와 2-패스 컴파일러의 장단점을 비교하여 설명하시오.

04 일반적으로 컴파일러를 구현할 때 고려해야 할 사항을 나열하고, 각각에 대해 간단히 설명하시오.

05 자신이 컴파일러를 만든다면 어떤 목적으로 어떤 특징을 가진 물리적 컴파일러를 만들 것인지 설명하시오.