



내공 있는 프로그래머로 길러주는

컴파일러의 이해

Chapter 08

중간 언어와 중간 코드 생성

목차

01 중간 언어

02 구문 지시적 번역

03 중간 코드 생성

학습목표

- 중간 언어의 개념과 종류에 대해 이해할 수 있다.
- 구문 지시적 번역에 대해 이해할 수 있다.
- 논리식과 치환문에 대한 중간 코드 생성을 한다.

■ 중간 언어의 개념

- **중간 코드(intermediate code)**는 구문 분석 단계에서 만들어진 구문 트리를 이용하여 생성되거나, 한 문법 규칙이 감축될 때마다 구문 지시적 번역으로 생성이 이뤄진다. 중간 코드를 생성하는 데는 **중간 언어(intermediate language, IL)**를 사용한다.
- 컴파일러 개발 과정을 여러 단계(multi-phase)의 모듈로 나누어 설계하게 되었고, 각 모듈을 연결해주는 정보를 저장하거나 표현할 수 있는 다양한 방법이 사용되었다. 그 중에서 컴파일러의 전단부와 후단부를 연결해주는 중간 코드에 대해 알아보자.

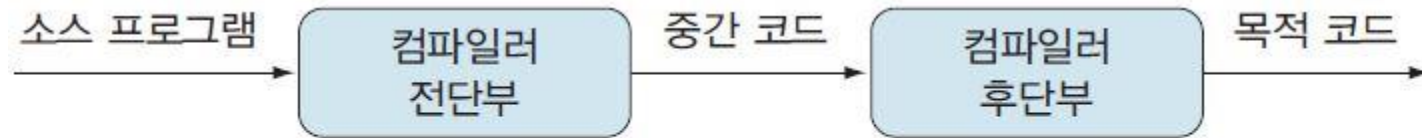


그림 8-1 중간 코드

- 컴파일러 개발 초기에는 프로그래밍 언어가 N개 있고 목적 기계가 M개 있다고 할 때 [그림 8-2]와 같이 $N \times M$ 개의 컴파일러 개발이 필요했으며, 이로 인한 개발 부담을 줄이기 위해 중간 코드의 중요성이 커졌다.

8.1 중간 언어

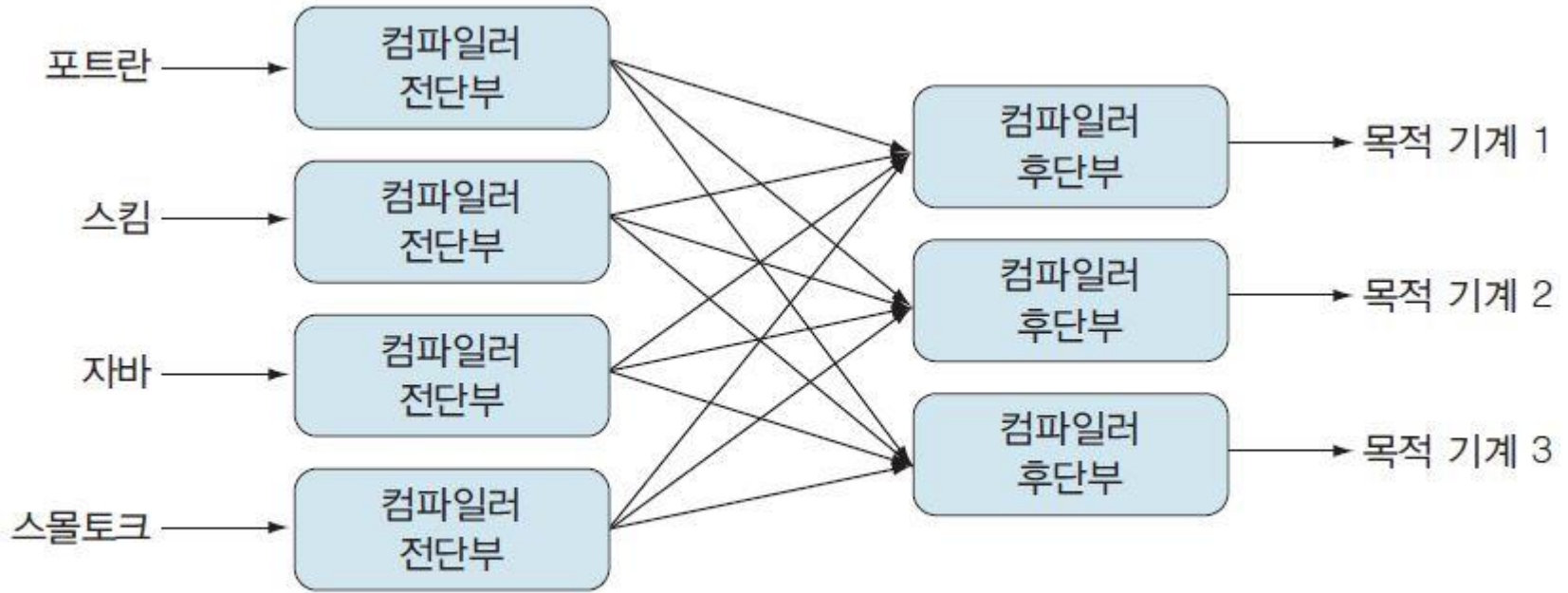


그림 8-2 중간 언어를 사용하지 않는 N×M개의 컴파일러 개발

- M개의 언어와 N개의 기계가 있을 때 필요한 컴파일러의 개수는 $M \times N$ 개이다. 하지만 UNCOL(universal compiler oriented language)의 개념으로 알려진 M개의 언어를 위한 전단부와 N개의 기계를 위한 후단부를 공통된 가상 기계(virtual machine, VM)가 연결할 수 있다면 이상적으로는 M개의 전단부 + N개의 후단부로 구성된 컴파일러 제작 시스템을 만들어 $M \times N$ 개의 컴파일러 대신에 $M + N$ 개의 컴파일러로 모두 해결할 수 있다.

8.1 중간 언어

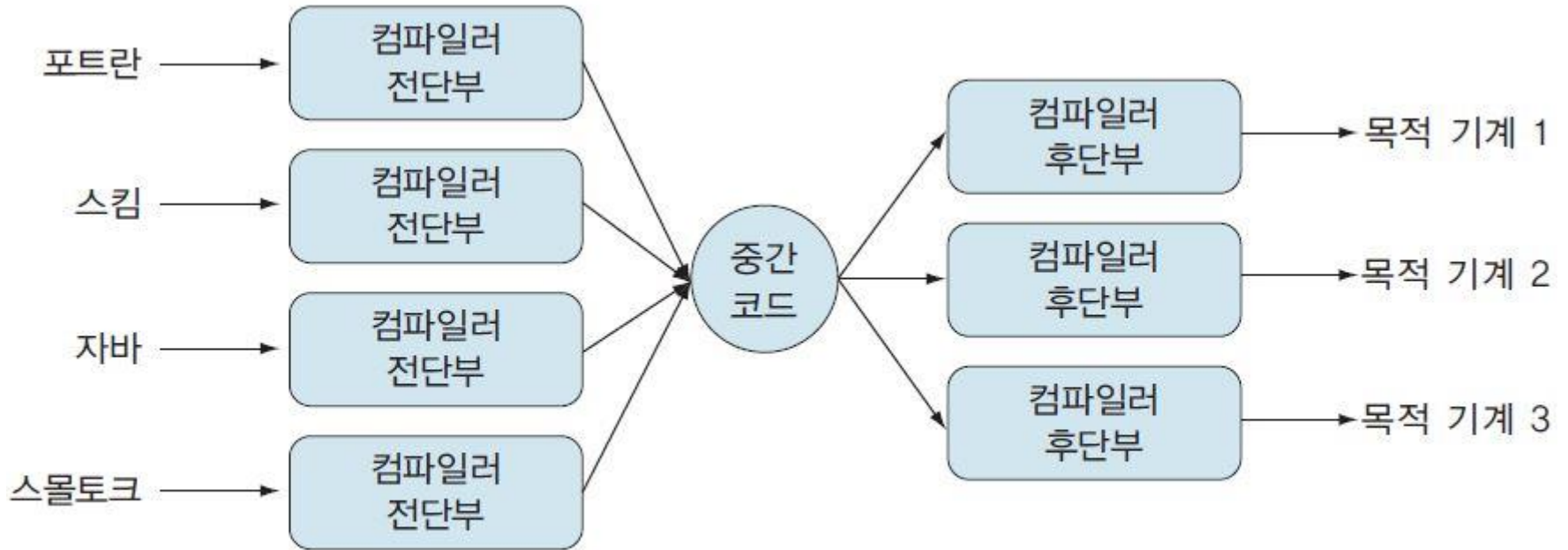


그림 8-3 중간 언어를 사용한 N + M개의 컴파일러 개발

■ IL 사용의 장점

- ① 소스 프로그램과 목적 기계의 특성에 독립적인 이식성 높은 컴파일러를 개발할 수 있고,
- ② 기계 독립적인 최적화를 수행하여 목적 기계 코드로 바뀐 상태에서 행하는 것보다 훨씬 효율적인 최적화 효과를 얻을 수 있으며,
- ③ 인터프리티브 컴파일링(interpretive compiling) 시스템에서 인터프리터를 이용하여 실행할 수 있다는 것이 장점이다.

■ IL사용의 단점

- 소스 프로그램을 직접 목적 기계 코드로 생성 하는 것보다 컴파일 시간이 더 많이 걸리고, 비효율적인 목적 기계 코드를 생성

■ 중간 코드를 생성하는 방법

- 중간 코드 생성기(intermediate code generator, ICG)를 이용하는 방법과 구문 지시적 번역(syntax-directed translation, SDT) 방법으로 나뉜다.

8.1 중간 언어

- 중간 코드 생성기는 소스 프로그램에 대한 구문 분석기의 출력을 입력으로 받아 중간 언어를 생성하는 컴파일 과정의 단계이다. 중간 코드 생성기는 파스 트리 또는 구문 트리를 순회하면서 효과적인 중간 코드를 생성한다.

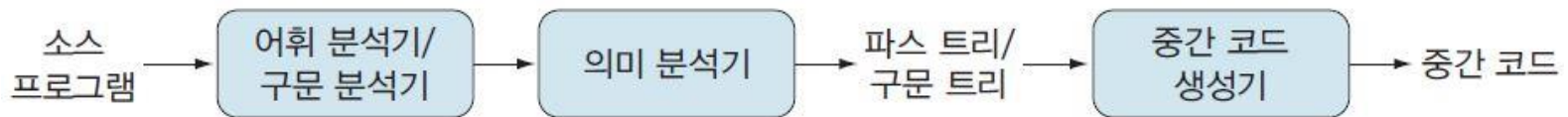


그림 8-4 중간 코드 생성

- 구문 지시적 번역은 문맥자유 문법으로 표현되는 각 생성 규칙에 대해 부 프로그램이나 의미 수행 코드를 기술하고, 의미 수행 코드는 구문 분석기에 의해 호출되어 실행된다.

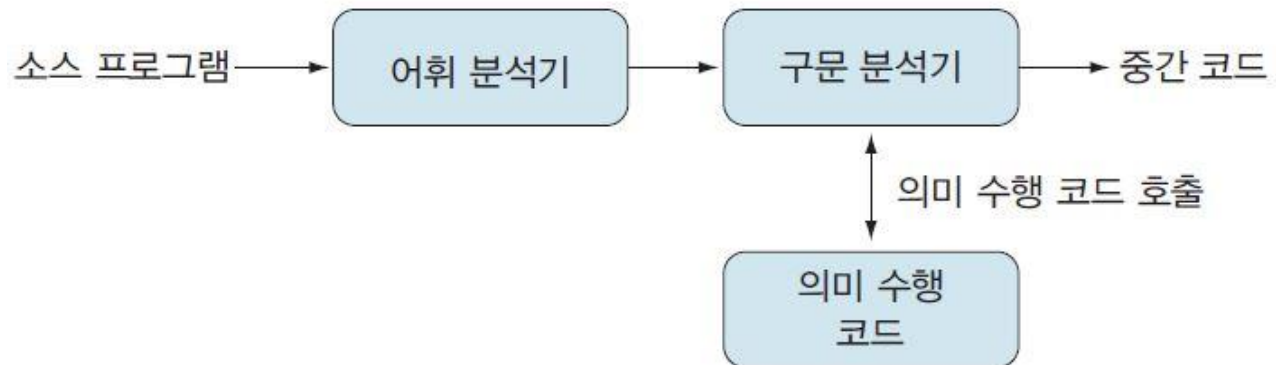


그림 8-5 구문 지시적 번역을 이용한 중간 코드 생성

- 중간 코드 생성기는 소스 프로그램에 대한 구문 분석기의 출력을 입력으로 받아 중간 언어를 생성하는 컴파일 과정의 단계이다. 중간 코드 생성기는 파스 트리 또는 구문 트리를 순회하면서 효과적인 중간 코드를 생성한다.

■ 중간 언어의 종류

- 폴란드식 표기법, 역폴란드식 표기법 → 전위 표기법, 후위 표기법
- 3-주소 코드 → Triple, 간접 Triple, Quadruple
- 트리 구조 코드 → AST, TCOL, Diana
- 가상 기계 코드 → P-코드, EM-코드, U-코드, 바이트코드(Bytecode)

■ 후위 표기법(역폴란드식 표기법)

- 중간 언어로 가장 오래된 것 중의 하나
- 포트란 컴파일러에서 산술식을 위한 중간 언어로 사용
- 인터프리터 언어인 베이직의 중간 언어로 사용
- 연산자의 위치에 따른 구분
- 산술식에서 피연산자가 연산자보다 먼저 나오는 형태

■ [예제 8-1] 후위 표기법으로 표현하기

- 산술식 $A = B * C + D / E$ 가 [그림 8-6]과 같이 구문 트리로 표현되어 있을 때 이를 후위 표기법으로 나타내보자.

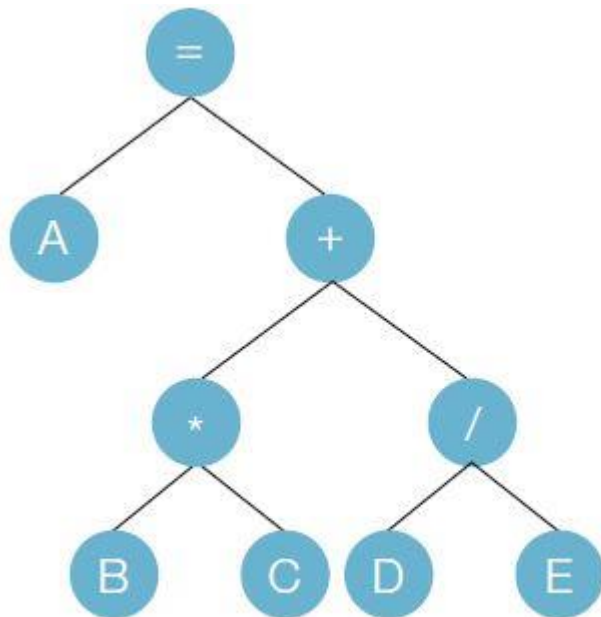


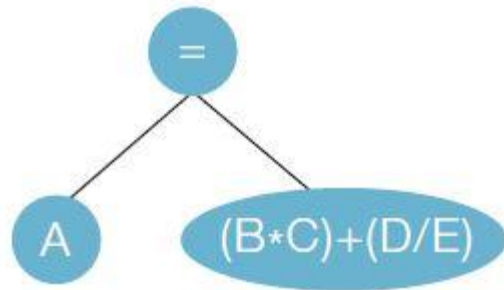
그림 8-6 $A = B * C + D / E$ 에 대한 구문 트리

- [풀이] 후위 표기법으로 나타내면 $ABC * DE / + =$ 이다.

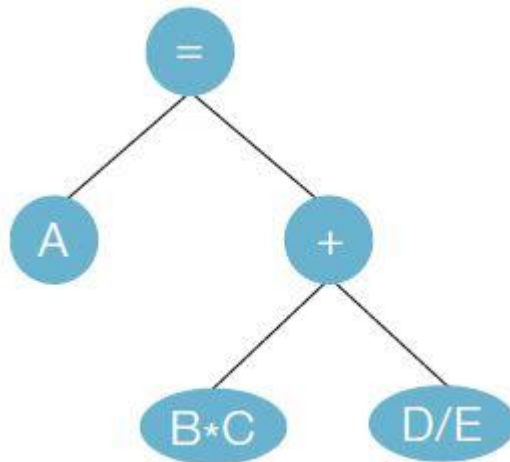
- 후위 표기법은 괄호를 사용하지 않고 식의 연산 순서가 임의로 정해지도록 표현할 수 있는 방법으로, 연산자 스택을 이용하여 쉽게 연산할 수 있고 소스 프로그램으로부터의 변환이 비교적 쉽고 빠르게 이뤄지기 때문에 인터프리터를 이용하는 언어의 중간 언어로 적합하다. 하지만 코드를 이동할 수 없으므로 최적화에는 부적합한 형태이다.
- 전위 표기법은 연산자를 피연산자 앞에 표현하는 형태로, 보편적으로 사용되지 않는 비교적 저급 수준의 중간 코드이다.
 - 이동-감축 파서를 이용하는데 파싱표의 상태수를 줄일 수 있음
- 1장에서 잠깐 언급했는데 이진트리를 선형으로 표기하는 폴란드식 표기법인 전위 표기법, 역폴란드식 표기법이라 불리는 후위 표기법, 그리고 중위 표기법에 대해 좀 더 자세히 살펴보자.
 - [그림 8-6]과 같이 치환문 $A = B * C + D / E$ 를 이진 트리로 표현해보자. 치환문을 이진 트리로 나타낼 때는 치환문에서 연산자의 우선순위와 결합 법칙을 이용하여 식의 계산 순서를 정한다. 연산자의 우선순위는 *와 /가 가장 높고 그 다음이 +와 -이다. 네 연산자 모두 왼쪽 결합 법칙을 취한다고 하자. 산술식의 결과 값을 치환하고 괄호를 이용하여 표현하면 다음과 같다.
 - $(A = ((B * C) + (D / E)))$

8.1 중간 언어

- 연산자 중에서 순위가 가장 낮은 연산자를 중심 연산자라 하는 데, 위 치환문에서 첫 번째 중심 연산자는 =이다. 중심 연산자를 부모 노드에 놓고 중심 연산자의 왼쪽 부분은 왼쪽 부분 트리, 중심 연산자의 오른쪽 부분은 오른쪽 부분 트리로 저장한다. 이러한 방법을 재귀적으로 수행하면 된다. 이를 적용하면 먼저 오른쪽과 같은 트리가 만들어진다.



- 오른쪽 부분 트리가 식이므로 다시 재귀적으로 처리한다. 식 $((B * C) + (D / E))$ 에서 중심 연산자가 +이므로 다음과 같은 트리가 만들어진다.



• 같은 방법으로 $B * C$ 와 D/E 를 트리로 표현하면 [그림 8-6]과 같아짐을 알 수 있다.

■ 이번에는 이진 트리로 표현된 치환문을 선형으로 나타내보자.

- 이진트리를 선형으로 표기하려면 이진 트리에서 각 노드를 방문하여 필요한 정보를 찾는 트리를 순회한다. 트리를 순회하는 방법은 전위 순회(preorder), 후위 순회(postorder), 중위 순회(inorder) 등 세 가지가 있다. 이때 전위 순회를 한 결과는 전위 표기법, 후위 순회를 한 결과는 후위 표기법, 중위 순회를 한 결과는 중위 표기법이라 한다. 전위 표기법은 연산자를 피연산자 앞에 놓는 표기 방법이고, 후위 표기법은 피연산자를 연산자 앞에 놓는 표기 방법이며, 중위 표기법은 피연산자와 피연산자 사이에 연산자가 오도록 표기하는 방법이다.
- 전위 순회는 루트, 왼쪽 부분 트리, 오른쪽 부분 트리를 재귀적으로 순회하는 방법이고, 후위 순회는 왼쪽 부분 트리, 오른쪽 부분 트리, 루트를 재귀적으로 순회하는 방법이다. 그리고 중위 순회는 왼쪽 부분 트리, 루트, 오른쪽 부분 트리를 재귀적으로 순회하는 방법이다. [그림 8-6]과 같은 이진트리에 대해 전위 순회를 한 결과인 전위 표기법은 $= A + * BC / DE$, 후위 순회를 한 결과인 후위 표기법은 $ABC * DE / + =$, 중위 순회를 한 결과인 중위 표기법은 $A = B * C + D / E$ 이다.

■ 3-주소 코드

- 코드의 재구성이 편리하기 때문에 코드 최적화에 적합
- 레지스터 기반 목적 기계 코드로의 변환이 편리하기 때문에 상용화 컴파일러에서 가장 많이 사용
- 치환 연산자의 오른쪽 부분이 1개의 단항, 이항 또는 논리 연산자와 피연산자로 구성되고, 치환 연산자의 왼쪽 부분은 식별자 또는 주소를 표현하여 하나의 코드 내에서 3개의 주소를 나타내는 형식의 중간 코드이다.
- 일반적인 3-주소 코드는 다음과 같다.
 - $A = B \text{ op } C$
 - 피연산자인 A, B, C는 식별자, 상수, 또는 컴파일러가 내부적으로 생성한 임시 변수이고, 연산자인 op는 단항 및 이항 산술 연산자 또는 논리 연산자이다.
- 이러한 3-주소 코드의 몇 가지 예
 - $A = B \text{ op } C$ 형식의 치환문 (여기서 op는 이항 산술 또는 논리 연산자이고, A는 결과 값을 저장하는 주소이며, B와 C는 이항 연산자 op의 피연산자이다.)
 - $A = \text{op } B$ 형식의 치환문(여기서 op는 단항 연산자이고, A는 결과 값을 저장하는 주소이며, B는 단항 연산자 op의 피연산자이다.)
 - $A = B$ 형식의 치환문
 - GOTO L 형식의 무조건 분기문

- 이러한 3-주소 코드의 몇 가지 예
 - $A = B \text{ op } C$ 형식의 치환문 (여기서 op 는 이항 산술 또는 논리 연산자이고, A 는 결과 값을 저장하는 주소이며, B 와 C 는 이항 연산자 op 의 피연산자이다.)
 - $A = \text{op } B$ 형식의 치환문(여기서 op 는 단항 연산자이고, A 는 결과 값을 저장하는 주소이며, B 는 단항 연산자 op 의 피연산자이다.)
 - $A = B$ 형식의 치환문
 - GOTO L 형식의 무조건 분기문(여기서 L은 레이블이며 L이 있는 3-주소 코드를 표시한다.)
 - if A relop B GOTO L과 같은 조건 분기문 (여기서 A와 B는 관계 연산자(relop)의 피연산자이며, 관계를 만족하면true 레이블 L에 대한 3-주소 코드를 실행하고 관계를 만족하지 않으면false 바로 다음 문장이 실행된다.)
 - Param A1
 - Param A2
 - ⋮
 - Param An
 - Call P, n
 - 위와 같은 프로시저 호출문
 - [여기서 n은 호출에서의 실 매개변수 개수이고 P는 프로시저의 이름에 해당된다. 이러한 코드는 호출 P(A1, A2, ..., An)의 3-주소 코드이다.]
 - $A = B[i]$, $A[i] = B$ 와 같은 형식의 첨자 치환문
 - $A = \&B$, $A = *B$, $*A = B$ 와 같은 형식의 주소 또는 포인터 치환문

▪ [예제 8-2] 3-주소 코드로 표현하기

- 치환문 $A = -B * (C + D)$ 를 3-주소 코드로 나타내보자.

- [풀이]

- $T_1 = -B$

- $T_2 = C + D$

- $T_3 = T_1 * T_2$

- $A = T_3$

- 3-주소 코드를 실제로 컴파일러에서 구현하기 위해 triple, quadruple, 간접 triple 표현 방법 등을 사용한다. triple 표현 방법은 결과 값을 저장하는 피연산자가 존재하지 않고, quadruple 표현 방법은 결과 값을 저장하는 피연산자가 존재한다. 또한 간접 triple 표현 방법은 triple 방법에서 코드 이동 시 발생하는 단점을 보완하기 위한 것이다.

■ Triple 표현 방법

- 하나의 기본 연산을 나타내기 위해 각 명령어가 <연산자>, <피연산자 -1>, <피연산자 -2>와 같은 3개의 필드로 구성된다.
- 임시 변수가 기호표에 들어가는 것을 피하기 위해 순서에 따라 번호가 부여되며, triple의 결과는 그 triple 번호에 할당된다.
- 이와 같이 triple의 결과를 triple 번호로 할당함으로써 [표 8-1]과 같이 오직 3개의 필드인 <op>, <피연산자 -1>, <피연산자 -2>가 있는 레코드를 구성할 수 있다.

표 8-1 [예제 8-2]에 대한 triple 표현

번호	연산자 <op>	<피연산자-1>	<피연산자-2>
[0]	uminus	B	
[1]	+	C	D
[2]	*	[0]	[1]
[3]	=	A	[2]

- 코드 최적화에서 흔히 발생하는 코드 이동 시에 triple을 지칭하는 triple 코드의 피연산자 또한 바뀌어야 하므로 triple 표현 방법은 최적화에 부적당하다.

■ 간접 Triple 표현 방법

- 코드 최적화에 부적당한 triple 표현 방법의 단점을 보완
- [표 8-2]와 같이 triple의 수행 순서를 별도의 표에 저장하여 코드 최적화 시에 표의 순서만 변경하고 원래의 triple을 변경하지 않음으로써 최적화를 보다 쉽게 할 수 있다.

표 8-2 [예제 8-2]에 대한 간접 triple 표현

수행 순서	번호	연산자 <op>	<피연산자-1>	<피연산자-2>
1.[0]	[0]	uminus	B	
2.[1]	[1]	+	C	D
3.[2]	[2]	*	[0]	[1]
4.[3]	[3]	=	A	[2]

■ quadruple 표현 방법

- [표 8-3]에서 보듯이 각 명령어가 <op>, <피연산자 -1>, <피연산자 -2>, <결과>와 같은 4개의 필드로 구성
- 여기서 <결과>는 <op> 연산의 결과를 저장하고 일반적으로 임시 변수가 사용되며, 실제로 목적 기계 코드 생성 시에 목적 기계의 레지스터 또는 메모리에 배정된다.
- quadruple 표현 방법은 triple 표현 방법과 달리 결과 주소 부분을 포함하고 있으므로 코드의 이동이 편리하여 최적화를 수행하는 데 적합하다.
- 하지만 많은 임시 변수를 관리해야 하는 단점이 있다.

표 8-3 [예제 8-2]에 대한 quadruple 표현

번호	연산자 <op>	<피연산자-1>	<피연산자-2>	<결과>
[0]	-	B		T ₁
[1]	+	C	D	T ₂
[2]	*	T ₁	T ₂	T ₃
[3]	=	T ₃		A

■ [예제 8-3] triple, 간접 triple, quadruple 표현 하기

- 다음 치환문을 triple, 간접 triple, quadruple로 표현해보자.
- $A = B + C * D / E$
- $F = C * D$
- [풀이]

triple		간접 triple		quadruple
[0]	$\langle *, C, D \rangle$	1. [0]	$[0] \langle *, C, D \rangle$	$\langle *, C, D, T_1 \rangle$
[1]	$\langle /, [0], E \rangle$	2. [1]	$[1] \langle /, [0], E \rangle$	$\langle /, T_1, E, T_2 \rangle$
[2]	$\langle +, B, [1] \rangle$	3. [2]	$[2] \langle +, B, [1] \rangle$	$\langle +, B, T_2, T_3 \rangle$
[3]	$\langle =, A, [2] \rangle$	4. [3]	$[3] \langle =, A, [2] \rangle$	$\langle =, T_3, A \rangle$
[4]	$\langle *, C, D \rangle$	5. [0]	$[4] \langle =, F, [0] \rangle$	$\langle *, C, D, T_4 \rangle$
[5]	$\langle =, F, [4] \rangle$	6. [4]		$\langle =, T_4, F \rangle$

■ 트리 구조 코드

- 소스 프로그램을 프로그램의 의미를 보다 시각적으로 표현할 수 있는 방법
- 코드의 특성상 손쉽게 재구성이 가능하므로 최적화 컴파일러의 IL로 가장 적합한 표현

■ 소스 프로그램의 트리 구조 표현 방법

▪ 파스 트리

- 구성하는 과정은 구문 분석(syntax analysis) 과정에서 문법-지시적(syntax-directed) 방법으로 손쉽게 이루어지나 불필요한 중복된 정보를 너무 많이 내포하고 있으므로 구문 분석을 하는 데 시간이 많이 걸리고 메모리의 낭비를 초래한다는 것 때문에 IL로는 부적당

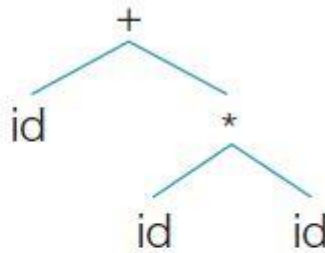
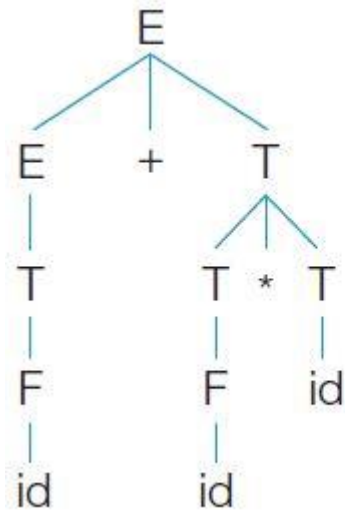
→ 효율적인 표현방법이 필요

→ 추상 구문 트리 (Abstract Syntax Tree)

- 파스 트리에서 의미 없는 논터미널을 제거하고 의미 있는 정보만을 트리 구조로 표현한 구문 트리를 사용하고 있다. 즉 구문 트리는 상수 또는 변수를 나타내는 터미널 노드와 연산자를 나타내는 논터미널 노드로 구성되며, 연산자의 피연산자들이 자식 노드로 연결되는 부분 트리를 구성한다.

■ [예제 8-4] 파스 트리와 구문 트리 구하기 1

- 다음 문법에서 입력 문자열 $id + id * id$ 에 대한 파스 트리과 구문 트리를 만들어보자.
- ① $E \rightarrow E + T$ ② $E \rightarrow T$
- ③ $T \rightarrow T * F$ ④ $T \rightarrow F$
- ⑤ $F \rightarrow (E)$ ⑥ $F \rightarrow id$
- [풀이]



■ 가상 기계 코드

- 최근에는 컴파일러를 소스 언어에만 관계된 전단부와 목적 기계에만 관계된 후단부로 분리 하고, 잘 정의된 인터페이스를 사용하여 이 사이를 연결함으로써 쉽게 이식되고 재목적 가능한 컴파일러를 제작하는 추세이다. 이와 같은 방법으로 만든 컴파일러를 재목적 컴파일러(retargetable compiler)라 하는데, 컴파일러를 한 기계에서 다른 기계로 이식하는 문제를 이렇게 후단부만 변경함으로써 쉽게 해결할 수 있다.
- 가상 기계는 컴퓨팅 환경을 소프트웨어로 구현한 것으로 컴퓨터를 에뮬레이션하는 소프트웨어이다.
- [그림 8-7]은 가상 기계를 이용한 재목적 컴파일러의 모델을 보여준다.

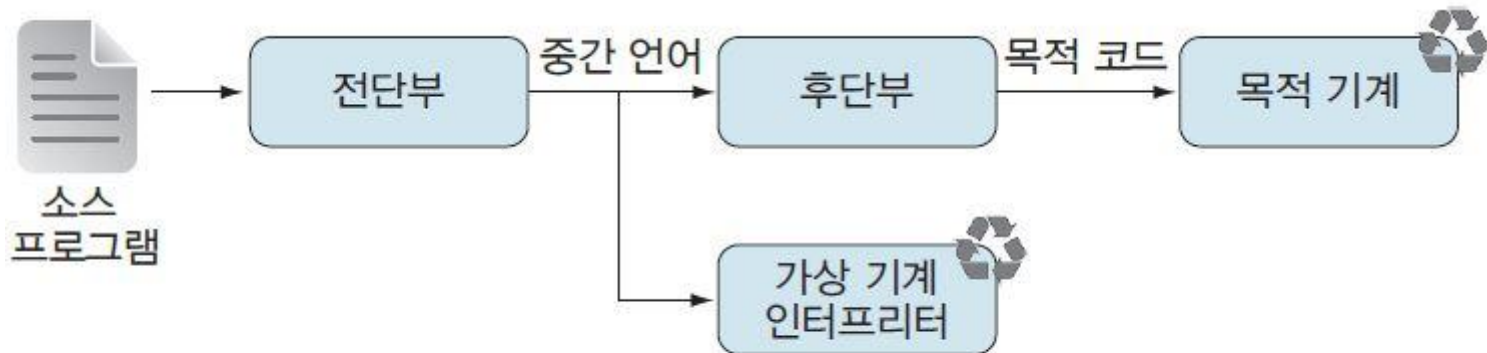


그림 8-7 재목적 컴파일러 모델

■ 가상 기계의 설계

- source language의 기본 연산과 모드에 근거
 - 기본 연산은 source program을 구성하는 가장 간단하고 대부분 직접적인 연산에 대응
 - 기본적인 자료 모드는 source language의 가장 간단한 형
 - 기본적인 모드와 연산들이 Abstract Machine의 명령어를 결정
- 실제 컴퓨터에 효율적으로 설치 가능 여부도 고려

- 이렇게 가상 기계를 이용하여 컴파일러를 만들면 전단부와 후단부가 명확히 구분되어 컴파일러의 이식성이 증가될 수 있다.
- 가상 기계에서 실행될 수 있는 명령어를 가상 기계 코드라고 한다. 가상 기계 코드는 필요에 의해 만들어진 여러 가지 종류가 있지만 여기서는 p-코드와 바이트코드에 대해 살펴보자.

■ P-코드

- p-코드는 컴퓨터에 별도의 컴파일러가 없어도 p-코드 인터프리터만 있으면 프로그램을 간단히 실행시킬 수 있다.
 - 하지만 목적 코드로 번역되는 한 단계가 추가되므로 실행 시간이 느리다는 것이 단점이다.
- p-코드는 소스 언어에서 p-코드로의 번역과 인터프리테이션이 비교적 쉽게 설계되었기 때문에, 파스칼-p 컴파일러는 p-코드에서 실제 기계로의 번역기 또는 p-코드 인터프리터를 사용함으로써 쉽게 이식할 수 있다.
- p-기계는 모든 연산이 스택에서 행해지는 스택 컴퓨터로 5개의 특수 레지스터와 메모리로 구성되어 있다.
- 메모리는 워드 단위로 된 선형 배열 형태로, p-코드 명령어가 저장되는 코드 부분과 자료가 저장되는 자료 부분으로 이뤄진다. [그림 8-8]은 p-기계의 메모리를 보여주며, p-기계의 메모리는 일반적인 메모리의 구조와 같다.



그림 8-8 p-기계의 메모리 구조

- 힙(heap)은 스택을 향해 커지며, 극단 포인터가 뉴 포인터보다 더 커지면 그 기계의 메모리가 모두 사용되었다는 것을 의미한다. 활성 레코드(activation record)라고도 하는 스택 프레임은 어떤 함수의 호출에 따라서 그와 관계되는 모든 자료를 저장해두는 스택 영역으로, 호출된 프로그램의 프레임이 스택 속에 순차적으로 쌓인다.
- 레지스터의 종류와 기능은 다음과 같다.
 - 프로그램 계수기program counter, PC : 코드 지역에서 현재 명령어의 위치를 가리킨다.
 - 스택 포인터stack pointer, SP : 스택의 톱을 가리킨다.
 - 마크 포인터mark pointer, MP : 활성 스택 프레임active stack frame의 시작을 가리킨다.
 - 극단 포인터extreme pointer, EP : 현재 사용할 수 있는 가장 큰 스택의 위치를 가리킨다
 - 뉴 포인터new pointer, NP : 힙의 톱을 가리킨다.

■ 바이트 코드

- 바이트코드는 자바 프로그래밍 환경에서 지원하는 중간 언어로 이기종 간의 실행 환경에 적합하도록 설계된 스택 기반의 가상 기계 코드
- 자바 컴파일러
 - 자바 프로그램을 입력으로 받아 플랫폼에 독립적인 자바 가상 기계(Java Virtual Machine, JVM) 상에서 동작하는 명령어 집합인 바이트코드를 생성해낸다.
 - 바이트코드는 이기종 간의 실행 환경에 적합하도록 설계되어 이식성과 유연성이 좋고, 인터프리터 방식과 JIT(just-in-time) 컴파일러 방식에 의해 실행되며, 확장자는 .Class이다. [그림 8-9]에 자바 가상 기계를 나타냈다.

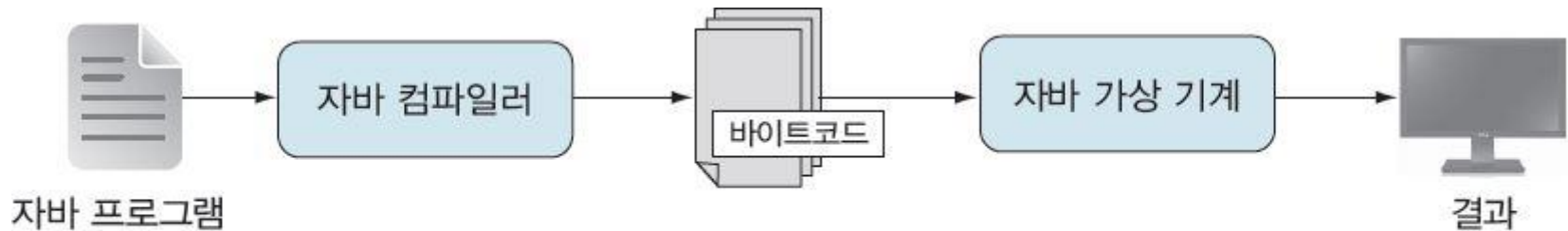


그림 8-9 자바 가상 기계

- 바이트코드에서 하나의 명령어 형식은 연산 코드(operation code) 부분과 연산 코드에 따라 여러 개의 피연산자 부분으로 구성된다. 현재 바이트코드는 230개의 명령어로 구성되어 있다. 바이트코드는 일반적인 가상 기계 코드의 특징을 모두 포함하고 있다.
- 바이트 코드만의 특징
 - 첫째, 네트워크상에서 전송 효율을 위해 코드를 가능한 한 작고 간결하게 설계한다.
 - 둘째, 자바 언어의 기능을 반영하는 명령어가 존재한다. 특히 배열, 클래스, 예외, 스레드 등을 직접 다룰 수 있는 명령어를 가지고 있다.
 - 셋째, 자료형에 따라 명령어가 각기 따로 존재한다. 예를 들어 iadd는 int 덧셈이고 fadd는 float 덧셈이다.
 - 넷째, 복합 기능을 가진 명령어를 제공 한다. 예를 들어 pop2는 스택에서 연속적으로 두 번을 삭제하는 기능을 한다.

8.1 중간 언어

▪ 중간 언어의 선택

중간 언어		역폴란드 식 표기법	3-주소 코드		트리 구조 코드	가상 기계 코드
			Quadruple	Triple		
평가 척도						
중간성 단계		C	B	B	C	B
효율적인 처리	소스 프로그램을 중간 언어로 번역	A	B	B	A	C
	중간언어를 목적 언어로 번역	C	B	B	C	A
	최적화	C	A	C	A	B
확장성		A	A	A	A	B

■ 구문 지시적 번역

- 각 생성 규칙에 해당하는 의미 수행 코드(semantic action code)를 직접 기술하여 필요한 일을 처리하는 방법이다. 따라서 문맥자유 문법의 각 생성 규칙은 컴파일러에 속하는 변수 값 계산, 중간 코드 생성, 에러 메시지 출력, 기호표에 정보 삽입 등과 관련된 부프로그램이나 의미 수행 코드로 구성되어 있다. 즉 구문 지시적 번역은 생성 규칙과 그에 해당하는 의미 규칙을 결합한 것이다.
- 구문 지시적 번역을 그림으로 나타내면 [그림 8-10]과 같다.

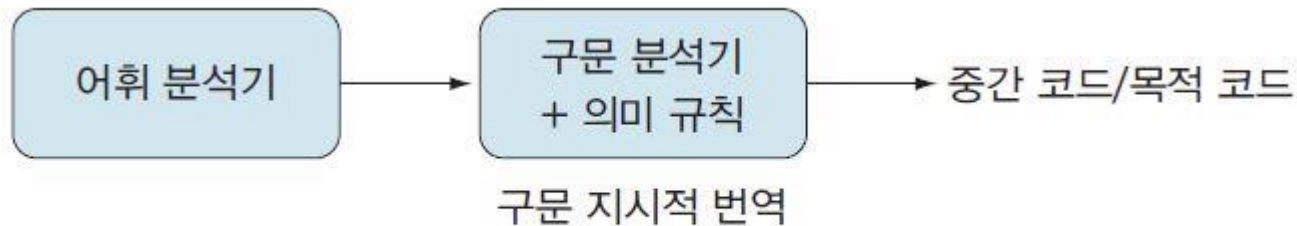


그림 8-10 구문 지시적 번역

- 다음과 같은 생성 규칙과 의미 규칙이 있다고 하자.
 - $E \rightarrow E1 + E2 \{E.VAL = E1.VAL + E2.VAL\}$
 - 여기서 의미 규칙은 중괄호 $\{ \}$ 로 묶어서 표시하고 생성 규칙 뒤에 나타냈다. 의미 규칙에서 생성 규칙의 왼쪽 부분과 관련된 번역 $E.VAL$ 은 오른쪽 부분의 E 와 관련된 번역 $E1.VAL$ 과 $E2.VAL$ 을 더하여 구하는 것이다. 앞의 생성 규칙에서 터미널 기호 $+$ 는 의미 규칙에 의해 수학적 의미로 번역.
- 구문 지시적 번역기의 구현
 - 구문 지시적 번역기를 만들기는 상당히 어려운데 여기서는 합성 속성만을 가지고 구문 지시적 정의를 이용하여 쉽게 만들어본다.
 - 합성 속성은 입력이 구문 분석될 때 상향식으로 계산된다. 감축이 발생할 때마다 감축되는 생성 규칙의 오른쪽 문법 기호에 대해 스택에 나타나는 속성을 가지고 새로운 합성 속성 값을 계산하는 것이다.
 - 상향식 구문 분석기는 구문 분석된 부분 트리에 대한 정보를 저장하기 위해 스택을 사용한다.

	STATE	VAL
VAL[Top] →	C	C.VAL
VAL[Top-1] →	B	B.VAL
⋮	A	A.VAL
	⋮	⋮

그림 8-11 스택을 이용한 구문 지시적 번역기의 구현

■ [예제 8-7] 구문 지시적 번역기 설계하고 수행하기

- [예제 7-1]과 [예제 7-2]의 구문 지시적 번역 방법과 주석 파스 트리를 이용하여 정수 피연산자와 +, * 연산자를 가진 계산기calculator 프로그램을 설계해보자. 그리고 설계된 계산기 프로그램에서 입력 문자열 23 * 5 + 4\$(입력의 마지막은 문자 \$로 표시)를 수행해보자.
- [풀이]

- 먼저 구문 지시적 번역기를 설계하기 위해 논터미널 기호 S, E, T, F를 사용하여 문법을 서술한다. 여기서 토큰 digit는 정수를 의미한다.
 - $S \rightarrow E\$$
 - $E \rightarrow E1 + T$
 - $E \rightarrow T$
 - $T \rightarrow T1 * F$
 - $T \rightarrow F$
 - $F \rightarrow (E)$
 - $F \rightarrow \text{digit}$
- 각 생성 규칙에 대한 의미 규칙을 나타내기 위해 논터미널 기호 E, T, F를 E.VAL, T.VAL, F.VAL 로 나타내고 토큰 digit의 변환은 LEXVAL로 나타낸다. LEXVAL의 값은 어휘 분석기가 제공한다고 가정한다. 각 생성 규칙에 대한 의미 규칙은 [예제 7-1]에 제시된 것과 같으며, 입력 문자열 $23 * 5 + 4\$$ 의 파스 트리는 [예제 7-2]의 그림과 같다.
- [표 8-4]는 계산기를 구현하기 위해 [예제 7-1]에 기술한 각 생성 규칙과 의미 수행 코드이다. 각 프로그램이 수행된 다음에는 새로운 스택의 톱이 NTOP으로 설정된다.

표 8-4 계산기 구현을 위한 생성 규칙과 의미 수행 코드

생성 규칙	의미 수행 코드
$S \rightarrow E\$$	print VAL[TOP]
$E \rightarrow E1 + T$	VAL[NTOP] = VAL[TOP-2] + VAL[TOP]
$E \rightarrow T$	
$T \rightarrow T1 * F$	VAL[NTOP] = VAL[TOP-2] * VAL[TOP]
$T \rightarrow F$	
$F \rightarrow (E)$	VAL[NTOP] = VAL[TOP-1]
$F \rightarrow \text{digit}$	VAL[NTOP] = LEXVAL

- **첨자** : 같은 문법 심벌이 생성규칙에 여러 번 나타나는 경우의 구별용
- **val** : 문법기호의 계산된 값을 기억하고 있는 속성
- **lexval** : 스캐너에서 받아온 토큰의 값
- [표 8-5]는 입력 문자열 $23 * 5 + 4\$$ 에 대한 구문 분석기의 수행 과정이다.
- [표 8-5]는 구문 분석기가 입력 $23 * 5 + 4\$$ 에 대해 어떻게 동작하는지, 그리고 스택의 STATE와 VAL 필드 항목이 각 동작 후 어떻게 바뀌는지를 보여준다. 여기서 STATE를 해당 문법 기호로 대치 하여 간결하게 표현했다. 또한 토큰 digit 역시 실제 입력 숫자로 대치했다.

8.2 구문 지시적 번역

표 8-5 $23 * 5 + 4\$$ 에 대한 구문 분석기의 수행 과정

입력 기호	STATE	VAL	적용된 생성 규칙
$23 * 5 + 4\$$			
$* 5 + 4\$$	23	23	
$* 5 + 4\$$	F	23	$F \rightarrow \text{digit}$
$* 5 + 4\$$	T	23	$T \rightarrow F$
$5 + 4\$$	T^*	$23_$	
$+ 4\$$	T^*5	23_5	
$+ 4\$$	T^*F	23_5	$F \rightarrow \text{digit}$
$+ 4\$$	T	115	$T \rightarrow T^*F$
$+ 4\$$	E	115	$E \rightarrow T$
$4\$$	E^+	$115_$	
$\$$	E^+4	115_4	
$\$$	E^+F	115_4	$F \rightarrow \text{digit}$
$\$$	E^+T	115_4	$T \rightarrow F$
$\$$	E	119	$E \rightarrow E^+T$
	$E\$$	$119_$	
	S	119	$S \rightarrow E\$$

- [표 8-5]의 첫 번째 동작에서 구문 분석기는 토큰 digit(속성 값은 23)에 해당하는 상태를 스택으로 이동한다. 두 번째 동작에서 구문 분석기는 생성 규칙 $F \rightarrow \text{digit}$ 로 감축하고 의미 규칙 $F.VAL = \text{digit.lexval}$ 을 수행시킨다. 세 번째 동작에서 구문 분석기는 $T \rightarrow F$ 로 감축한다. 그런데 이 생성 규칙에 연관된 코드가 없기 때문에 VAL의 값이 변하지 않는다.
- 다음으로 *를 읽고 나서 5를 읽는다. 다음 동작에서 구문 분석기는 생성 규칙 $F \rightarrow \text{digit}$ 로 감축하고 의미 규칙 $F.VAL = \text{digit.lexval}$ 을 수행시킨다. 그 다음 동작으로 구문 분석기는 $T \rightarrow T * F$ 로 감축하므로 115라는 값이 VAL에 저장된다. 그리고 구문 분석기는 $E \rightarrow T$ 로 감축하는데 이때도 VAL 값이 변하지 않는다. 그런 다음 +를 읽고 나서 4를 읽는다. 구문 분석기는 생성 규칙 $F \rightarrow \text{digit}$ 로 감축하고 의미 규칙 $F.VAL = \text{digit.lexval}$ 을 수행시킨다. 다음 동작으로 구문 분석기는 $E \rightarrow T$ 로 감축하는데 이때도 VAL 값이 변하지 않는다. 그 다음 동작으로 구문 분석기는 $E \rightarrow E + T$ 로 감축하므로 119라는 값이 VAL에 저장된다. 그런 다음 \$를 읽으며 마지막으로 $S \rightarrow E\$$ 로 감축하고 119를 출력한다.

- 중간 코드를 생성하는 방법은 구문 지시적 번역에 의해 구문 분석 과정에서 중간 코드를 생성하는 방법과, 파스 트리 또는 구문 트리와 같은 중간 표현을 생성 한 후 이로부터 중간 코드를 생성하는 방법으로 나눌 수 있다.
- 구문 지시적 번역 방법의 경우
 - 소스 프로그램에 구문 에러(syntax error)가 발생하면 에러 발생 전까지 수행한 중간 코드 생성이 무의미해지므로 시간 낭비를 초래하며, 생성 규칙의 감축 순서에 따라 중간 코드를 생성하므로 구현이 복잡해지는 단점이 있다. 하지만 구문 분석 과정에서 직접 중간 코드를 생성하므로 소스 프로그램에 구문 에러가 없는 경우에는 오히려 빠르게 중간 코드를 생성할 수 있다는 것이 장점이다.
- 소스 프로그램에 대해 중간 표현을 생성한 후 중간 코드를 생성하는 방법
 - 중간 코드 생성에 필요한 의미 있는 정보만을 파스 트리 또는 구문 트리에 저장하고 있으므로 코드 생성기의 구현이 매우 용이하지만, 컴파일러의 크기가 커지고 컴파일 시간이 길어진다.

■ 논리식

- 논리식(부울식, boolean expression)은 논리 값을 계산하거나, if 또는 while 문에서 조건식을 계산하는 데 사용된다. 논리식은 and, or, not과 같은 논리 연산자와 부울 변수이거나 관계식으로 표현되는 피연산자로 구성된다. 논리식을 생성하는 다음 문법을 생각해보자.
 - $E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id relop id} \mid \text{true} \mid \text{false}$
- 여기서 relop는 $<, \leq, =, \neq, >, \geq$ 와 같은 6개의 관계 연산자이고, 논리 연산자 or와 and는 왼쪽 결합 법칙을 만족하며, 연산자 우선순위는 or가 가장 낮고 다음은 and, not 순서라고 가정한다.
- if 문이나 while 문 등은 논리식 구현에 적합하다. 예를 들어 $E1 \text{ or } E2$ 에서 $E1$ 이 true라면 $E2$ 를 검사할 필요 없이 전체 식의 값이 true가 되는 단락 회로(short circuit) 평가 방법을 사용한다.
- 만약 언어의 정의에 의해 논리식의 한 부분을 검사하지 않아도 된다면 컴파일러는 논리식의 계산에서 하나의 표현식만을 계산하게 함으로써 최적화할 수 있다.
- 논리식을 3-주소 코드로 번역하기 위해 true는 1로, false는 0으로 하고 왼쪽 결합 법칙을 사용하는 논리식의 변환을 생각해보자. 예를 들어 논리식 $A \text{ or } B \text{ and not } C$ 에 대한 3-주소 코드는 논리 연산자의 우선순위가 not, and, or이므로 다음과 같다.

- $T1 = \text{not } C$
- $T2 = B \text{ and } C$
- $T3 = A \text{ or } T2$
- 관계식 $A < B$ 는 조건문 `if A < B then 1 else 0`과 일치하므로 3-주소 코드로 변환하면 [그림 8-12]와 같다.

```
① if A < B GOTO ④  
② T = 0  
③ GOTO ⑤  
④ T = 1  
⑤
```

그림 8-12 조건문 `if A < B then 1 else 0`에 대한 3-주소 코드

- 또한 논리식 $A < B \text{ or } C$ 에 대한 3-주소 코드는 [그림 8-14]와 같다.

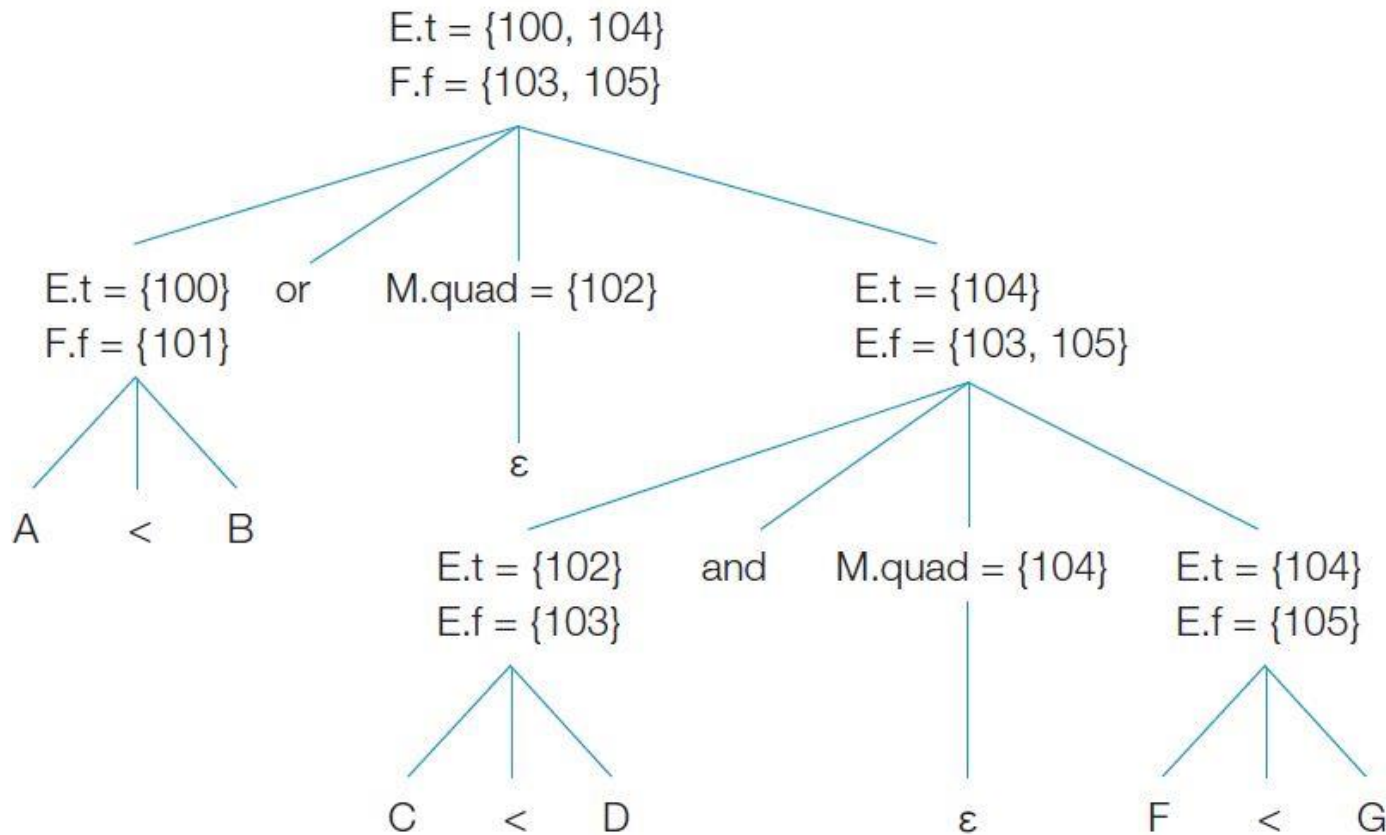
```
① if A < B GOTO ④  
②  $T_1 = 0$   
③ GOTO ⑤  
④  $T_1 = 1$   
⑤  $T_2 = T_1 \text{ or } C$ 
```

그림 8-14 $A < B \text{ or } C$ 에 대한 3-주소 코드

- ① $E \rightarrow E_1 \text{ or } E_2$ {backpatch(E_1 .falselist, M.quad);
 E .truelist = merge(E_1 .truelist, E_2 .truelist);
 E .falselist = E_2 .falselist}
- ② $E \rightarrow E_1 \text{ and } E_2$ {backpatch(E_1 .falselist, M.quad);
 E .truelist = E_2 .truelist);
 E .falselist = merge(E_1 .falselist, E_2 .falselist)}
- ③ $E \rightarrow \text{not } E_1$ { E .truelist = E_1 .falselist;
 E .falselist = E_1 .truelist}
- ④ $E \rightarrow (E_1)$ { E .truelist = E_1 .truelist;
 E .falselist = E_1 .falselist}
- ⑤ $E \rightarrow \text{id}_1 \text{ relop id}_2$
 { E .truelist = makelist(nextquad);
 E .falselist = makelist(nextquad+1);
 emit('if' id_1 .place relop id_2 .place 'GOTO-');
 emit('GOTO-')}
- ⑥ $E \rightarrow \text{true}$
 { E .truelist = makelist(nextquad);
 emit('GOTO-')}
- ⑦ $E \rightarrow \text{false}$
 E .falselist = makelist(nextquad+1);
 emit('GOTO-')}
- ⑧ $M \rightarrow \epsilon$ { M .quad = nextquad}

■ [예제 8-8] 주석 파스 트리와 quadruple 코드 만들기

- 다음 논리식에 대한 주석 파스 트리를 만들고, 백패칭을 이용하여 quadruple 코드로 만들어보자.
- $A < B$ or $C < D$ and $F < G$
- [풀이] 먼저 주석 파스 트리는 다음과 같다.



- 처음 문장 번호를 100번으로 가정한 경우, 논리식의 구문 분석은 다음과 같은 순서로 진행된다.
- 1. 첫 번째 구문 분석은 논리식 $A < B$ 로부터 $E \rightarrow id1 \text{ relop } id2$ 의 의미 분석에 의해 $E.t = \text{makelist}(\text{nextquad})$ 를 수행한다. $\text{nextquad} = \{100\}$ 이므로 $E.t = \{100\}$ 으로 결정된다. 또한 $E.f = \text{makelist}(\text{nextquad}+1)$ 이므로 $E.f = \{101\}$ 로 결정된다. quad 는 3-주소 코드로 표현된 순차 구조 형식으로 보면 더 이해하기 쉽고 시작 위치를 결정하는 문장의 위치로 간주할 수 있다. 따라서 다음과 같은 문장이 생성된다.
- 100 if A < B GOTO _
- 101 GOTO _
- 또한 $M.\text{quad} = \{102\}$ 가 된다.
- 2. 두 번째 구문 분석은 논리식 $C < D$ 로, $M.\text{quad} = \{102\}$ 이므로 $E.t = \{102\}$, $E.f = \{103\}$ 이고 $M.\text{quad} = \{104\}$ 가 된다. 그리고 다음과 같은 문장이 생성된다.
- 102 if C < D GOTO _
- 103 GOTO _

- 3. 세 번째 구문 분석은 논리식 $F < G$ 로, $M.quad = \{104\}$ 이므로 $E.t = \{104\}$, $E.f = \{105\}$ 가 된다. 그리고 다음과 같은 문장이 생성된다.
 - 104 if $F < G$ GOTO _
 - 105 GOTO _
- 이제까지 생성된 6개의 문장을 나열하면 다음과 같다.
 - 100 if $A < B$ GOTO _
 - 101 GOTO _
 - 102 if $C < D$ GOTO _
 - 103 GOTO _
 - 104 if $F < G$ GOTO _
 - 105 GOTO _
- 4. 다음 순서는 $C < D$ and $F < G$ 로, $E \rightarrow E1$ and $M E2$ 의 의미 분석 규칙에 의해 $backpatch(E1.t, M.quad)$ 를 실행한다. 이는 곧 $E1.t = \{102\}$ 이므로 102에 있는 문장의 분기 목표가 $M.quad$ 가 되어 다음과 같이 된다.
 - 102 if $C < D$ GOTO 104
- 이어서 $E.t = E2.t$ 이므로 $E.t = \{104\}$ 가 된다. 또한 $E.f = Merge(E1.f, E2.f)$ 이므로 $E.f = \{103, 105\}$ 이다. 따라서 다음과 같은 문장이 생성된다.

- 100 if A < B GOTO _
- 101 GOTO _
- 102 if C < D GOTO 104
- 103 GOTO _
- 104 if F < G GOTO _
- 105 GOTO _
- 5. 다음 순서는 $A < B$ or M E로, $E \rightarrow E1$ or M E2의 의미 분석 규칙에 의하면 $\text{backpatch}(E1.t, M.\text{quad})$ 에 따라 $E1.f = \{101\}$, $M.\text{quad} = \{102\}$ 이므로 다음과 같이 생산된다.
- 101 GOTO 102
- 그리고 $E.f = \{103, 105\}$, $E.t = \{100, 104\}$ 이므로 $E2.f = \{103, 105\}$ 이다. 결국 다음과 같이 된다.
- 100 if A < B GOTO _
- 101 GOTO 102
- 102 if C < D GOTO 104
- 103 GOTO _
- 104 if F < G GOTO _
- 105 GOTO _