제1장 컴파일러 개요

1.1 컴퓨터 시스템과 컴파일러와의 관계

- 컴퓨터의 간단한 구조와 CPU 동작

1. 메모리에 특정 주소에 저장된 값을 레지스터의 값과 연산(+, -, *, /) 하라.
2. 메모리의 어느 주소에 있는 내용을 CPU의 레지스터로 이동하라.
3. CPU와 메모리 사이에서 정보를 이동하라.
4. 비교하는 인스트럭션 등

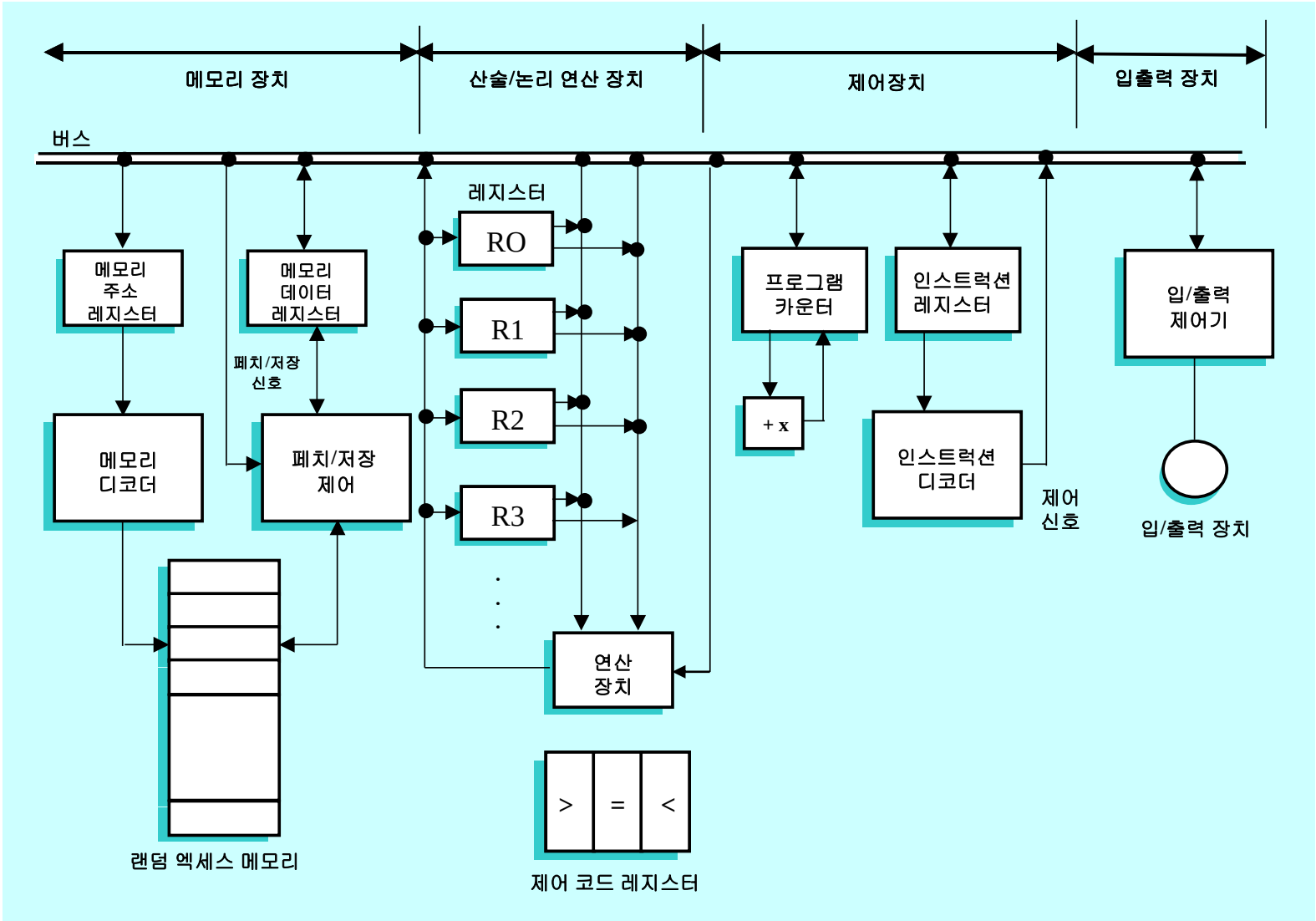

그림 1.1 컴퓨터 구조 블록 다이어그램

어셈블리어 프로그램메모리에 저장

```
program test(input, output);
int a, b, c, k;
float t,y,x,s,r;
{
  a = 3;
  b =4;
  k =a +b;
}
```

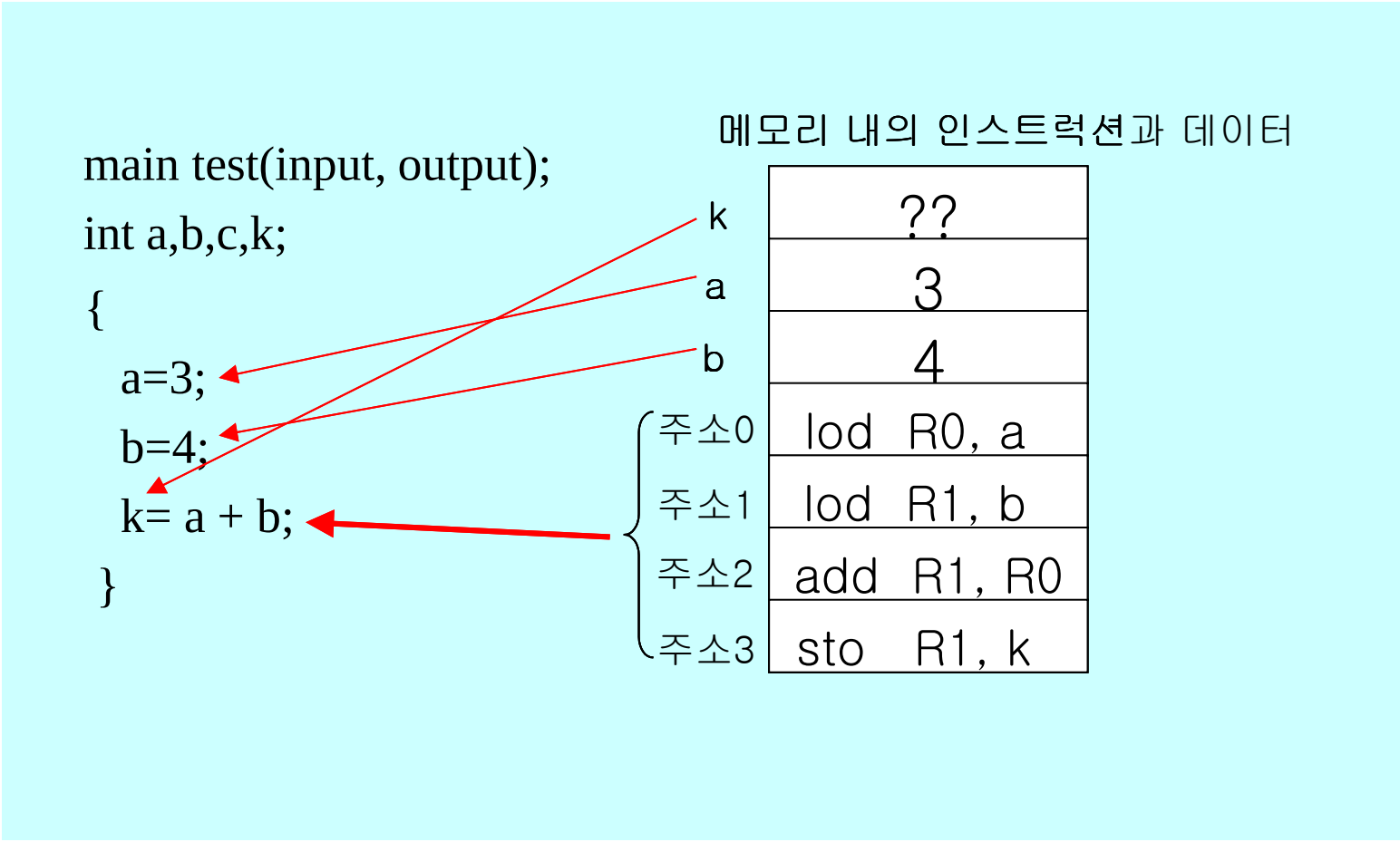

그림 1.2 C 로 작성한 프로그램 예를 번역한 메모리 내의 어셈블리어

의미

1. 메모리의 a 주소의 내용(3)을 레지스터 R0로 이동
2. 주소 b의 내용을(4)를 레지스터 R1으로 이동
3. R0와 R1을 더하여 R1에 저장
4. 그 후 R1의 내용(7)을 주소 k로 이동하여 저장

컴파일러

- 고급 프로그래밍 언어의 문장을 읽어서 기계어로 된 여러 인스트럭션으로 번역한다.
- 고급언어로 작성된 Y = Y + 7 과 같은 문장을 처리하려면, 컴파일러가 직접 “더하기” 연산을 수행하는 것이 아니라 “더하기”를 인스트럭션과 실행에 필요한 여러 인스트럭션을 생성한다.
- 예

```
lod   R0, Y
add   R0, =‘7’
sto   R0, Y
```

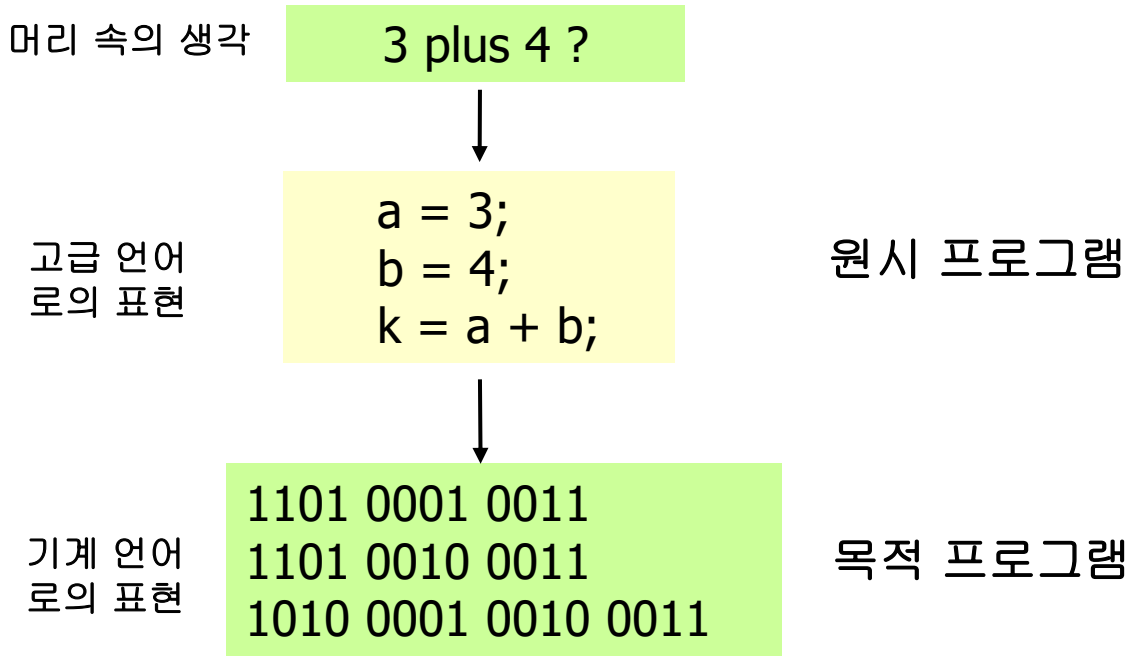

그림 1.3 프로그래밍 언어 수준의 번역 예

언어 번역 레벨

- 컴파일러는 특정 언어(L_1)를 입력으로 하여 변환된 다른 언어(L_0)를 결과로 출력하는 프로그램이다.
  - L_1은 원시언어이며 L_0는 컴파일러 C가 L_1을 번역한 목적 언어

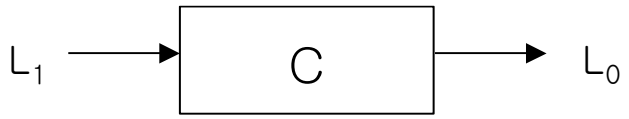

그림 1.4 원시 언어 L_1을 목적 언어 L_0 로 변환하는 컴파일러

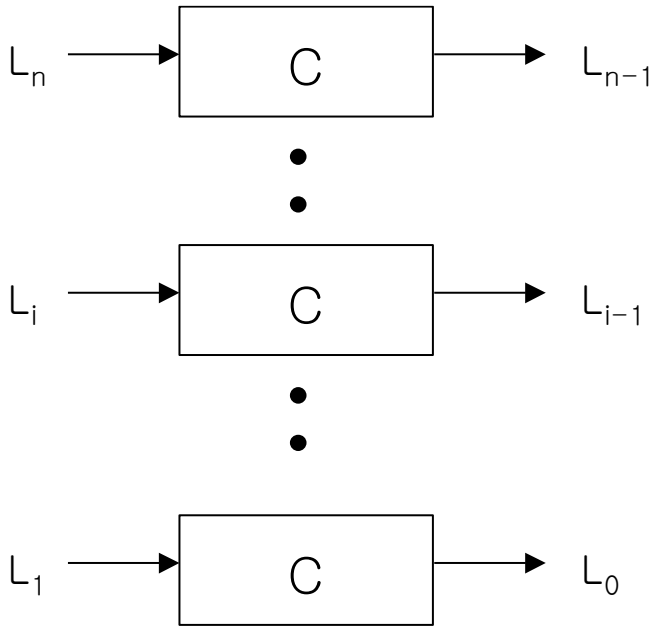

그림 1.5 어떤 언어를 다른 목적 언어로 번역하는 컴파일러

수식 번역 예

- A = B * C + D 와 같은 C 프로그래밍 언어의 할당문은 다음과 같이 어셈블리어 인스트럭션으로 번역한다.

```
lod   R1, B       /* B의 값을 레지스터 1 에 넣어라          */
mul   R1, C       /* C의 값과 레지스터 1 의 값을 곱하라     */
sto   R1, TMP1    /* TMP1에 결과를 저장하라                 */
lod   R1, D       /* D의 값을 레지스터 1 에 넣어라          */
add   R1, TMP1    /* TMP1의 값을 레지스터 1 에 더하라       */
sto   R1, TMP2    /* TMP2에 결과를 저장하라                 */
mov   A, TMP2     /* TMP2를 A로 옮겨라. 최종결과이다.       */
```

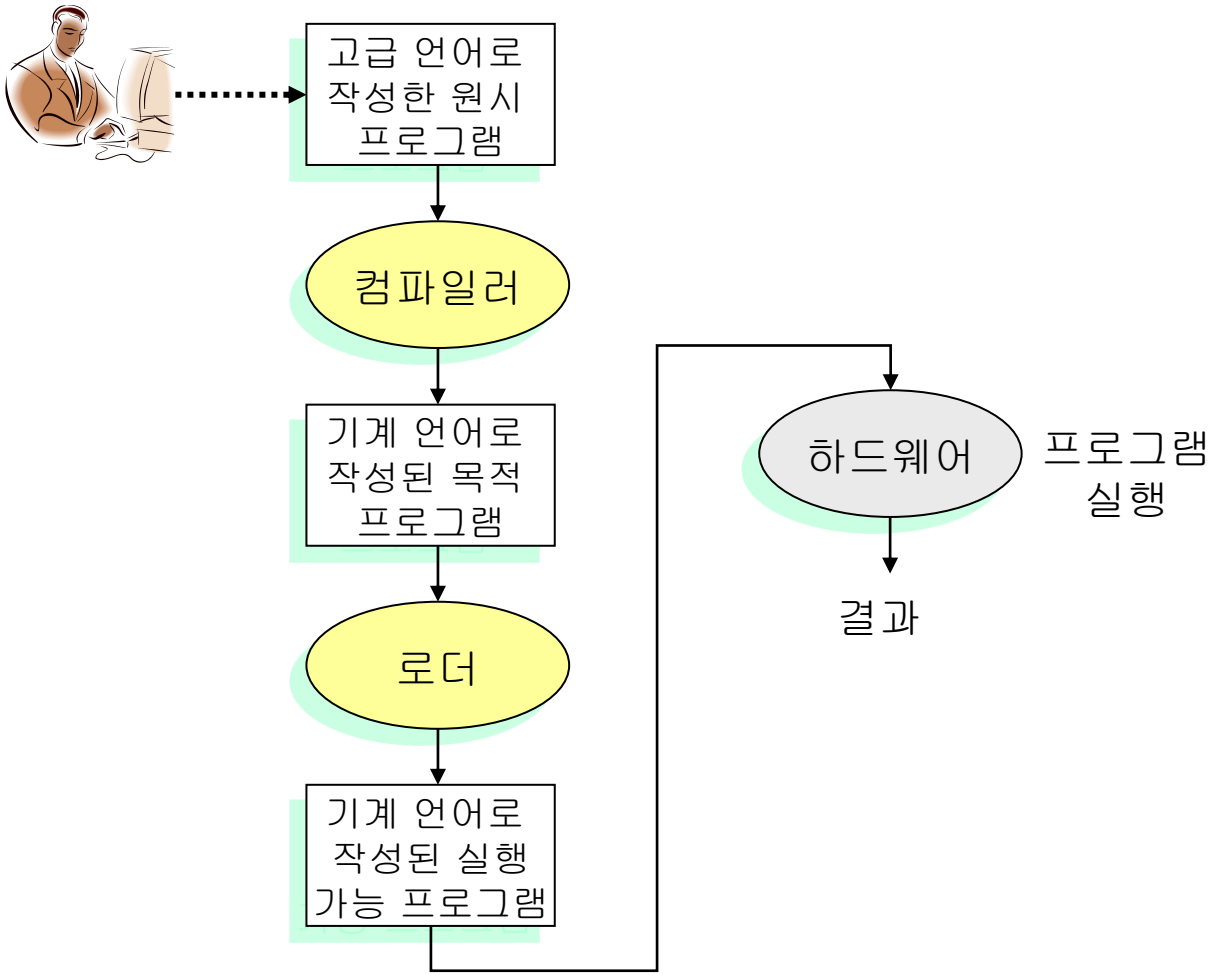

그림 1.6 원시 프로그램이 컴퓨터 시스템에서 번역,탑재,실행 과정

에러의 예 관찰하기

```
  ......
S1:  X = 0;
S2:  phi = 3.14;
S3:  B = (X * Y + 120;
S4:  6 = B * B * 3.14;
S5:  if (B > A) B= B -1 else B = A;
S6:  Y = Y / X * 2 * phi * R;
S7:  value = Y * p;   /* 변수  p는 프로그램에 선언되지 않았다고 가정
     .....
```

결과

- 문장 S3과 S4 는 컴파일 시간에 발견되는 에러이다.
- 문장 S3에서는 왼쪽 괄호에 대응하는 괄호인 ‘)’ 가 빠졌다.
- 문장 S4에서는 할당문에 상수가 와서 어떤 값을 저장할 수 없다. 즉, 할당문의 왼쪽에는 변수가 있어야 한다.
- 문장 S6에서는 변수 Y 를 X 로 나누는데 X 값은 이미 문장 S1에서 0 으로 할당되어 있어서 변수 X로 나누기를 할 수 없다(계산 불능).
- 문장 S7에서는 p 값이 선언되어 있지 않아서 실행 중 지정한 주소를 발견할 수 없으므로 실행 에러가 발생한다.

- 실습 1.1 아래 C 언어로 작성한 원시 프로그램 문장을 어셈블리어 인스트럭션으로 변환하시오.

```
k = a * b + c;
```

- 실습 1.2 다음과 같이 C 언어로 작성한 원시 프로그램 문장에 대하여 컴파일러의 출력인 어셈블리어 인스트럭션을 보이시오.

```
c = (a -b) * k - (a + b);
```

컴파일러와 인터프리터

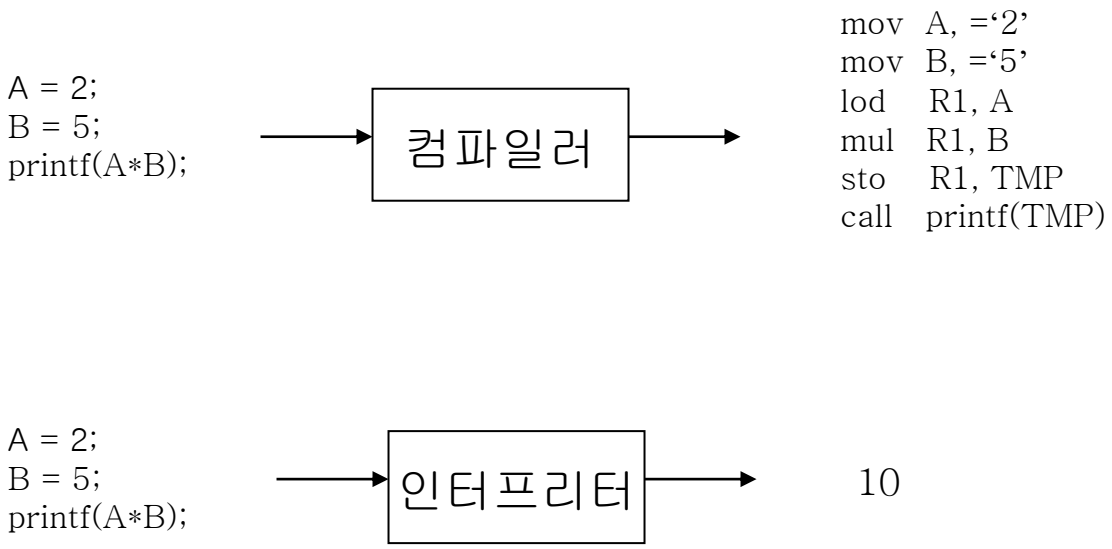

그림 1.7 같은 입력에 대한 컴파일러와 인터프리터의 실행 결과

- 실습 1.3 아래 고급 언어로 작성한 문장을 컴파일한 결과를 보이시오.

```
a = 3;
b = 4;
c = 5;
a = a * b + c;
printf(a);
```

- 실습 1.4 아래 고급 언어로 작성한 문장의 인터프리터 출력 결과를 보이시오.

```
a = 2;
b = 7;
c = 3;
a = a * (b + c);
printf(a);
```

1.2 컴파일러 구조

- 어휘 분석 단계
- 구문분석 단계
- 코드 생성 단계
- 코드 최적화 단계

컴파일러의 간단한 구성과 각 단계

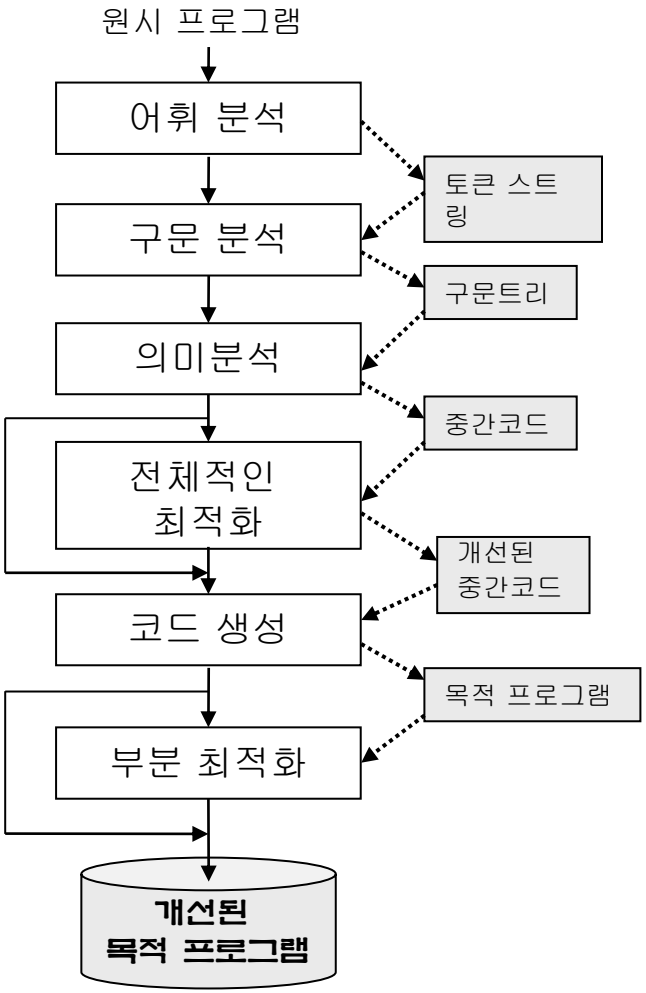

그림 1.8 컴파일러의 간단한 구성과 각 단계

1.2.1 어휘 분석:프로그램 문장에서 단어를 분류

- 어휘 분석 단계
- 토큰 예

```
main test(input, output);
int a, b, c, k;
{
  a = 3;
  b =4;
  k =a + b;
}
```

빈칸을 기준으로 각 문자와 단어 분리

```
main test ( input , output ) ; int a , b , c ,
k ; { a = 3 ; b = 4 ; k = a + b ; }
```

표 1.1 분류한 토큰의 예

분리한 단어(토큰)	속성
main input output int	예약어(의미를 가짐)
test a b c k	식별자(함수이름, 변수)
(,) ; { = }	분리문자
3 4	상수(정수형)
+	연산자(+)

표 1.2 고급 언어에서의 토큰의 종류

속성	토큰 예	의미
예약어	main, function, while, if, for, array, int, float, case, switch, do, ...	실행 코드 생성이나 자료형등의 정보를 제공
식별자	test, a, tax, sum, ...	사용자가 선언한 값들, 즉, 변수나 함수 이름
연산자	+, *, –, /, %, ...	수식 연산 코드생성
상수	134, 15, 0.054–E3, 345.56, –23.3, ...	상수와 정수
문자	a, A "KNU", ...	단일 문자나 따옴표(“ ”) 안의 문자 스트링
분리문자	. ( ) , ; : ...	다른 문자와 구별하는 문자들
주석	// 스트링, { 스트링 }, /* 스트링 */	프로그램에 다는 설명문, 프로그램의 실행에는 영향을 주지 않는다.

심볼 테이블

표 1.3 위 프로그램에서 사용한 심볼 테이블 예

식별자	속 성	
test	사용자가 선언한 함수 이름=test	메모리의 위치 포인터
a	이름=a,정수형, 16비트, 단일 변수	메모리의 위치 포인터
b	이름=b,정수형, 16비트, 단일 변수	메모리의 위치 포인터
c	이름=c,정수형, 16비트, 단일 변수	메모리의 위치 포인터
k	이름=k,정수형, 16비트, 단일 변수	메모리의 위치 포인터

- 실습 1.5 아래 C 프로그래밍 언어의 문장에 대하여 어휘 분석한 결과인 토큰과 속성을 나타내시오.

```
ave = sum / student;
grade = rate * (ave + score)/0.3;
```

1.2.2 구문 분석

- **프로그램에 있는 문장을 문법과 대조하기**
- **파싱**이라고도 한다.
- 문법에 따라 원시 프로그램의 기본 구조를 파악하여 문법 구조에 따라 **신택스 트리**를 생성한다.
  - 원시 프로그램의 각 문장에 대하여 문법구조를 나타낸다.
  - 신택스 트리에서 중간코드를 생성할 수 있다.
  - 신택스 트리에서 중간 노드는 연산자나 제어 구조를 나타내고, 잎 노드는 피연산자를 나타낸다.

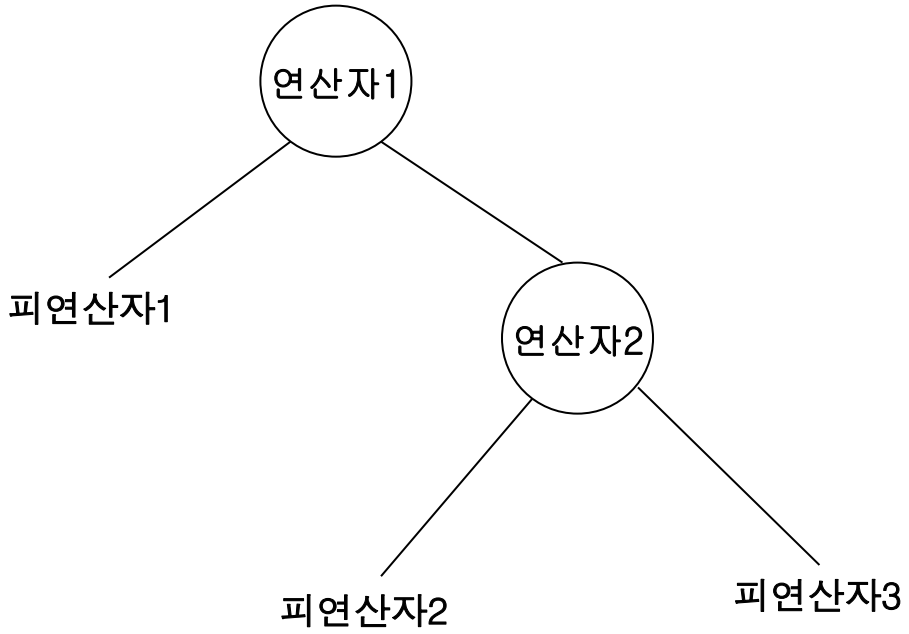

그림 1.9 산술 수식의 신택스 트리

A = B + C * D 의 신택스 트리 예

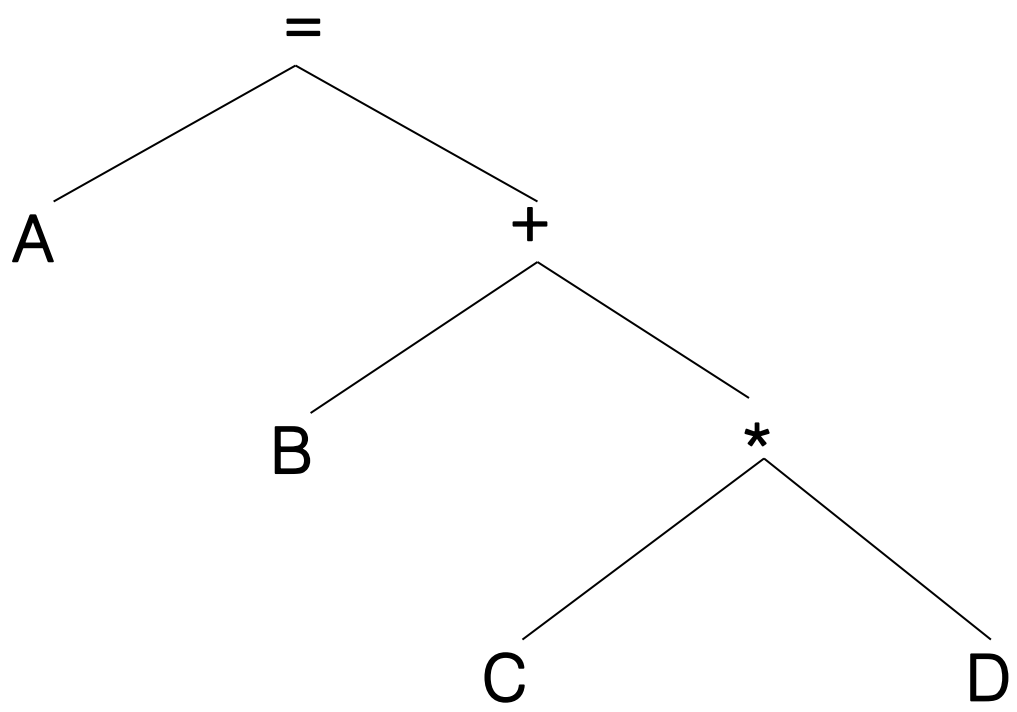

그림 1.10 A = B + C * D 의 신택스 트리

제어문의 신택스 트리

- if 제어 문장은 세 개의 자식 노드가 있는 신택스 트리로 구성된다.

```
if (불리언 수식)
    {
     stmt1
    }
  else
    {
     stmt2
    }
```

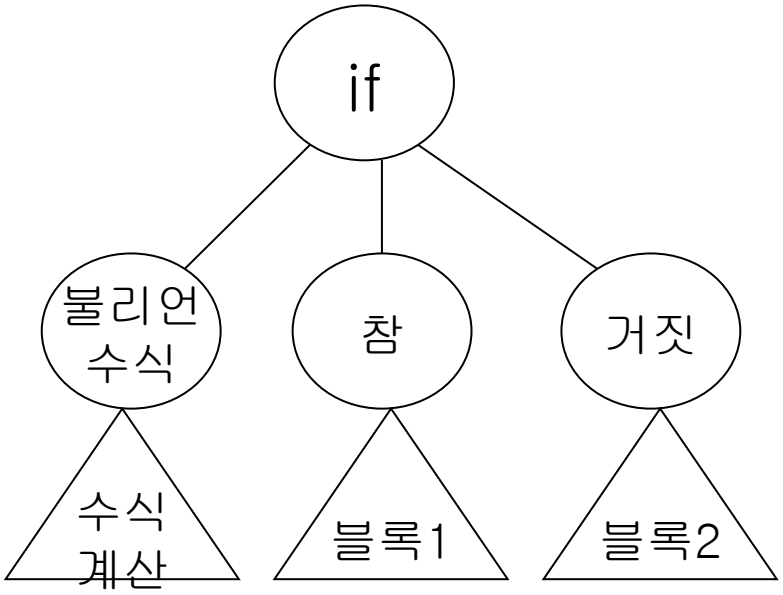

그림 1.11 제어 문장 if 에 대한 신택스 트리

WHILE 문의 신택스 트리

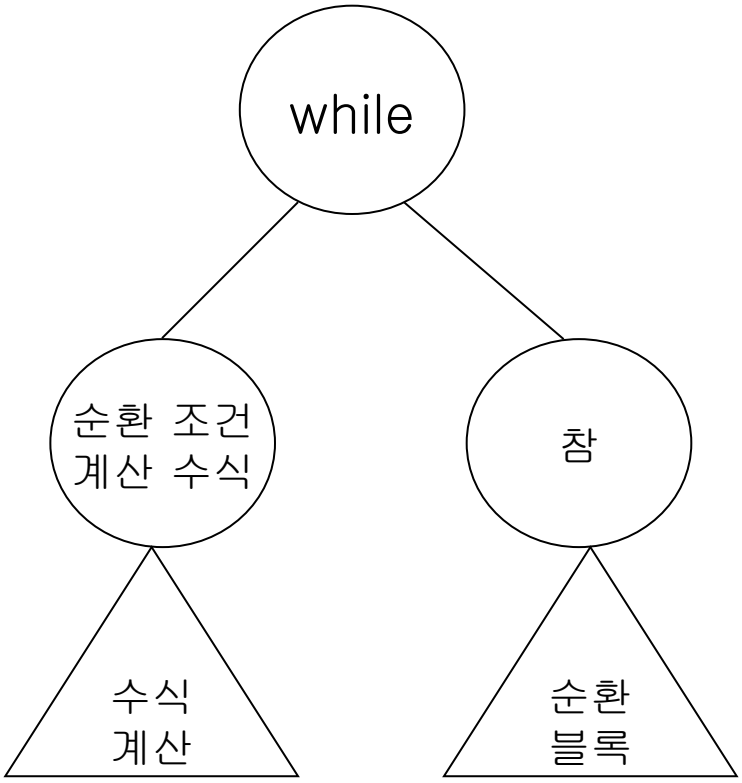

그림 1.12 제어 문장 while 에 대한 구문 트리

- 실습 1.6 아래 C 문장에 대한 신택스 트리를 보이시오.

```
if (flag_bit == 1)
    {
      cur_addr = cur_addr + 1;
     }
  else
    {
      reset_bit = 0;
     }
```

1.2.3 의미 분석

- **변수에 자료형을 할당하고 중간코드 생성하기**
- 의미 분석 단계에서는 변수나 식별자에 대한 **자료형**을 할당
- 목적 코드 생성을 위한 중간 코드를 생성한다.
  - 중간코드는 목적 컴퓨터의 기계어 인스트럭션으로 대응할 수 있다.
  - 중간 코드 예

```
add  A, B, C
```

  - 메모리 주소 A의 값과 주소 B의 값을 더하여 그 결과를 주소 C에 저장하라

- 실습 1.7 아래 C 원시 입력에 대하여 중간코드를 생성하시오.

  A = B * C + D

레이블

- 다음 실행할 중간코드의 '주소'가 L1이라고 지정

(LBL L1)

- 실습 1.8아래 C 문장에 대한 중간코드를 보이시오.

```
score = score * 0.3;
if (score >= 90) grade = 'A'
        else
            grade = 'C';
```

1.2.4 전역 코드 최적화

- 코드 최적화 기법
  - 전역 코드 최적화
  - 지역 코드 최적화

최적화 예

```
L1:    ....
       (comp    SCORE,  le, AVE, L2)  ----------(1)
       (mov      ='A', GRADE)
       (jmp      L3)    ----------------------(2)
       (mul     T1,  B, C)
       (add     T2,  D, T1)
       (mov     T2, A)
L2:
       (mul      SCORE, SCORE, =0.3)
       (jmp      L1)
L3:    ....
```

최적화 결과

```
L1:  ....
    (comp  SCORE,  le, AVE, L2)
    (mov    ='A', GRADE)
    (jmp    L3)
L2:
    (mul    SCORE, SCORE, =0.3)
    (jmp    L1)
L3:  ....
```

1.2.5 코드 생성

- **코드 생성**에서는 중간코드에서 목적 컴퓨터에 맞는 어셈블리어나 기계어를 생성한다.

```
lod  R1, A   (두 번째 피연산자 A의 내용을 레지스터 R1에 넣어라)
add R1, B   (두 번째 피연산자 B의 내용을 레지스터 R1의 값과 더
              하고 결과를 R1에 저장하라)
sto  R1, C   (레지스터 R1의 내용을 C 에 저장하라)
```

- 실습 1.9 아래 중간 코드에서 피연산자가 두 개인 인스트럭션(어셈블리어 형식)을 보이시오.

```
(add   Temp1, A, B )
(mul   Temp2, Temp1, Temp1)
(mov  Temp2,  B)
```

1.2.6 지역 코드 최적화

- **지역 코드 최적화**는 기계에 종속된 최적화

- K = A + B * C 의 목적 프로그램 코드

```
lod  R1, B     { 주소 B의 내용(값)을 레지스터 R1으로 이동}
mul R1, C      { 주소 C의 내용(값)을 레지스터 R1과 곱하고 결과를 R1에 저장}
sto R1, T1     { 곱해진 결과인 R1의 내용(값)을  주소 T1에 임시로 저장}
lod R1, T1     { 주소 T1의 내용(값)을 레지스터 R1로 이동}
add R1, A      { 주소 A의 내용(값)을 레지스터 R1의 내용(값)과 더하고 결과
                 를 R1에 저장}
sto R1,  K     { 더해진 결과인 R1의 내용(값)을  주소 K에 저장}
```

지역 코드 최적화 결과

```
lod   R1, B   { 주소 B의 내용(값)을 레지스터 R1으로 이동}
mul   R1, C   { 주소 C의 내용(값)을 레지스터 R1과 곱하고 결과를 R1에 저장}
add   R1, A    { 주소 A의 내용(값)을 레지스터 R1의 내용(값)과 더하고 결과
                 를 R1에 저장}
sto   R1,  K   { 더해진 결과인 R1의 내용(값)을  주소 K에 저장}
```

1.3 컴파일러 설계와 구현

- 혼자서 설계하고 구현하기
- 컴파일러-컴파일러 도구로 구현하기

1.3.1 혼자서 설계하고 구현하기

- 혼자 구현하려면 어휘 분석 단계에서 필요한 이론인 형식언어, 정규 수식, 정규 문법, 유한 상태 기계 등을 알아야 한다.
  - 구문 분석 단계에서는 정규 문법과 언어, LR 파싱 이론, LR 아이템, 파싱 테이블 등
  - 의미 분석 단계에서는 중간 코드를 생성하는 액션 루틴, 속성 문법 이론
  - 최적화 단계에서는 프로그램 제어와 자료 흐름 분석 이론
  - 코드 생성에서는 컴퓨터의 구조, 운영체제, 레지스터 할당 등 이론

1.3.2 컴파일러-컴파일러 도구로 설계하고 구현하기

- 자동으로 컴파일러를 만드는 **컴파일러-컴파일러**
- **lex**와 **yacc**
  - lex는 어휘 분석기를 생성하는 유틸리티
  - yacc는 구문 분석기를 생성하는 유틸리티

- lex 를 이용하여 쉽게 개발할 수 있는 소프트웨어
  - 워드 프로세서의 수학공식이나 수식을 표현하는 도구
  - 특정 언어로 작성한 소프트웨어를 다른 언어로 변환하는 도구
  - 데이터베이스 질의어
  - 자연어 번역기
  - 휴대폰 같은 모바일 단말기의 프로세서를 위한 최적화 컴파일러 등

제 1 장

끝