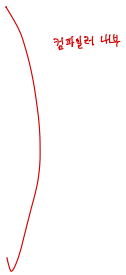
프로그래밍언어 구현

Programming language implementation

목차

- Source program to machine code
  - Preprocessor
  - Compiler
  - Assembler
  - Linker
- Compiler, interpreter, hybrid
- Compiler 내부 단계
  - 어휘 분석
  - 구문 분석
  - 의미 분석
  - 중간 코드 생성



Compiler, assembler, interpreter

High-level programming language(고수준 프로그래밍 언어)
- C++, Java, Python 등과 같이 높은 수준의 추상화(abstraction)를 통해 기계의 속성을 숨긴 프로그래밍언어

Low-level programming language(저수준 프로그래밍 언어)
- 기계(CPU 명령어, 레지스터, 어드레싱 모드 등) 사용에 대한 추상화를 거의 혹은 전혀 적용하지 않은 프로그래밍 언어로 assembly language(어셈블리어) 및 machine language(기계어)가 있음

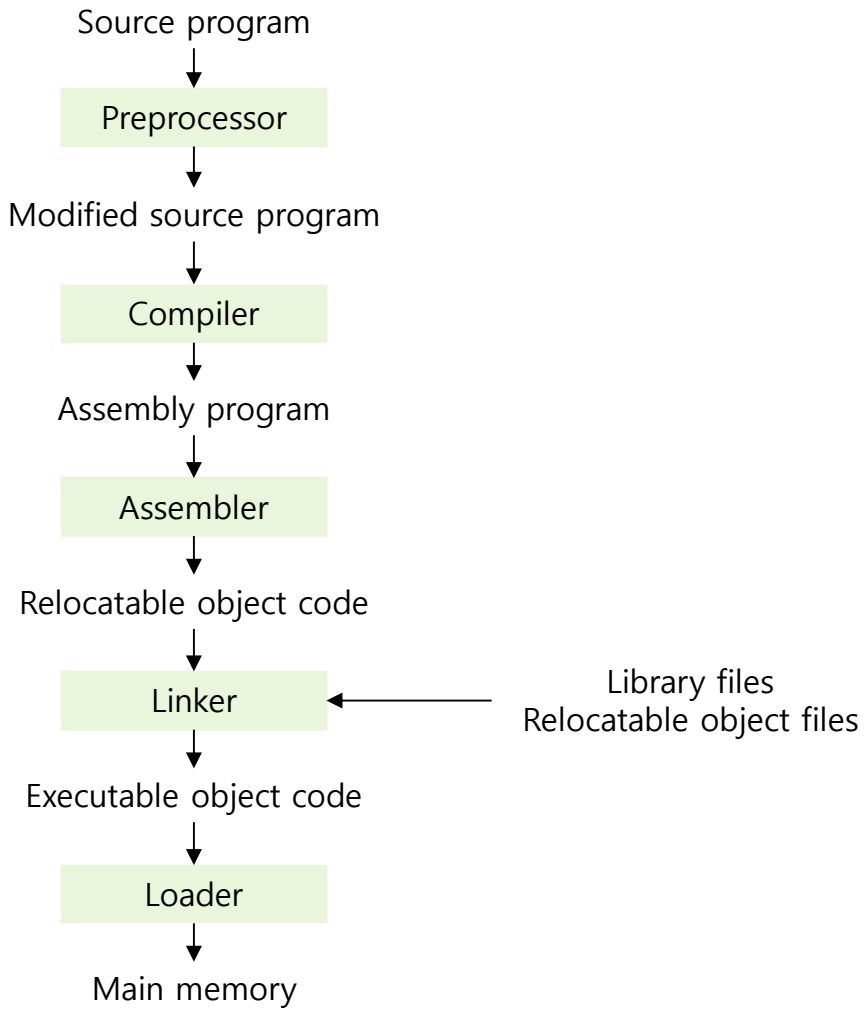
프로그래밍언어구현(programming language implementation)
- 프로그래밍언어를 실행하는 시스템(예: 컴파일러, 해석기)

Compiler(컴파일러)
- 한 프로그래밍 언어(source language, 원시 언어)로 작성된 코드를 다른 프로그래밍 언어(target language, 목적 언어)로 변환하는 프로그램
- 고수준 프로그래밍 언어로 작성된 코드를 저수준 프로그래밍 언어로 번역하는 프로그램

Assembler(어셈블러)
- 어셈블리어로 작성된 코드를 기계어 코드로 변환하는 프로그램

Interpreter(해석기)
- 한 프로그래밍 언어로 작성된 코드를 (기계어 코드로 변환하지 않고) 직접 실행하는 프로그램

Source program to machine code



Preprocessor (1/2)

컴파일러
-고수준 언어로 작성된 프로그램을 저수준 언어로 변환
C => 어셈블러

전처리기(preprocessor)
- 컴파일이 수행되기 전에 소스코드를 변환(주석 제거, 매크로 확장, 파일 내용 포함, 조건부 변환 등)하는 프로그램
- C 언어 preprocessor 프로그램은 cpp

util.c

```
/*
 * 2024.03.02
 */
#define PI 3.14
#define max(x,y) (((x)>(y))?(x):(y))
// areaOfCircle()
double areaOfCircle(double r){
    return PI*r*r; // return areaOfCircle
}

// ft1()
int ft1(int n1, int n2){
    return max(n1,n2)+1;
}

// ft2()
int ft2(int p, int q){
    return 2*max(p,q)+PI;
}
```

매크로 이름

cpp util.c -o util.i

혹은

gcc -E util.c -o util.i

util.i

linemarkers

```
# 0 "util.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "util.c"




double areaOfCircle(double r){
 return 3.14*r*r;
}


int ft1(int n1, int n2){
 return (((n1)>(n2))?(n1):(n2))+1;
}


int ft2(int p, int q){
 return 2*(((p)>(q))?(p):(q))+3.14;
}
```

매크로(macro)
- 매크로는 이름이 부여된 코드 조각으로 **#define** 지시자(directive)를 사용하여 정의됨
- 정의된 매크로 이름은 사용될 때마다 전처리기에 의해 매크로 내용으로 교체(확장)됨
- Object-like macro와 function-like macro가 있음
- 함수형 매크로는 매크로 정의 시 매크로 이름 직후 괄호 필요

Preprocessor (2/2)

util.h

```
#ifndef UTIL_H
#define UTIL_H
int f(int x, int y);
int g(int x, int y);
int h(int x, int y);
#endif
```

util.h

```
int f(int x, int y);
int g(int x, int y);
int h(int x, int y);
```

#include "util.h"가 여러 번 쓰여도 실제 한번만 include되게

C header file
- 여러 소스 파일 간에 공유되어야 할 선언과 매크로 정의 등을 포함하는 파일(C 헤더파일은 관례상 .h로 끝남)

#include 지시자
- 헤더파일의 내용은 C 전처리기 지시자 #include를 통해 소스코드에 포함됨

#include <filename>
- 시스템 표준 디렉토리 (및 명령행 옵션 -I에 명시된 디렉토리)에서 **filename**에 해당하는 파일을 찾아 그 내용을 소스코드에 포함시킴

#include "filename"
- 현재 소스파일의 디렉토리에서 **filename**에 해당하는 파일을 찾아 그 내용을 소스코드에 포함시키고, 실패 시 #include <filename>의 동작 수행

#ifndef UTIL_H
- 이전에 UTIL_H가 정의되지 않았으면 #endif 직전까지 내용을 수행 (조건부 변환)

#define UTIL_H
- 매크로 UTIL_H를 정의(매크로 정의 시 body는 optional)

util.c

```
#include "util.h"
int f(int x, int y){
   return 2 * h(x,y);
}
int g(int x, int y){
   return 1 + f(x,y);
}
int h(int x, int y){
   return x + y;
}
```

cpp →

util.i

```
int f(int x, int y);
int g(int x, int y);
int h(int x, int y);
int f(int x, int y){
   return 2 * h(x,y);
}
int g(int x, int y){
   return 1 + f(x,y);
}
int h(int x, int y){
   return x + y;
}
```

main.c

```
#include <stdio.h>
#include "util.h"
int main(void){
    int x=f(3,4);
    int y=g(3,4);
    int z=h(3,4);
    printf("%d %d %d", x,y,z);
    return 0;
}
```

cpp →

main.i

```
...
int printf(const char *format, ...);
...
int f(int x, int y);
int g(int x, int y);
int h(int x, int y);
int main(void){
 int x=f(3,4);
 int y=g(3,4);
 int z=h(3,4);
 printf("%d %d %d", x,y,z);
 return 0;
}
```

실행 (-P 옵션 사용 시 linemarker 제거)

```
cpp -P util.c -o util.i
cpp -P main.c -o main.i
```

Preprocess, compile, assemble

util.c

```
#define BASE 100
int ft(int x){
    return BASE+x;
}
```

util.i

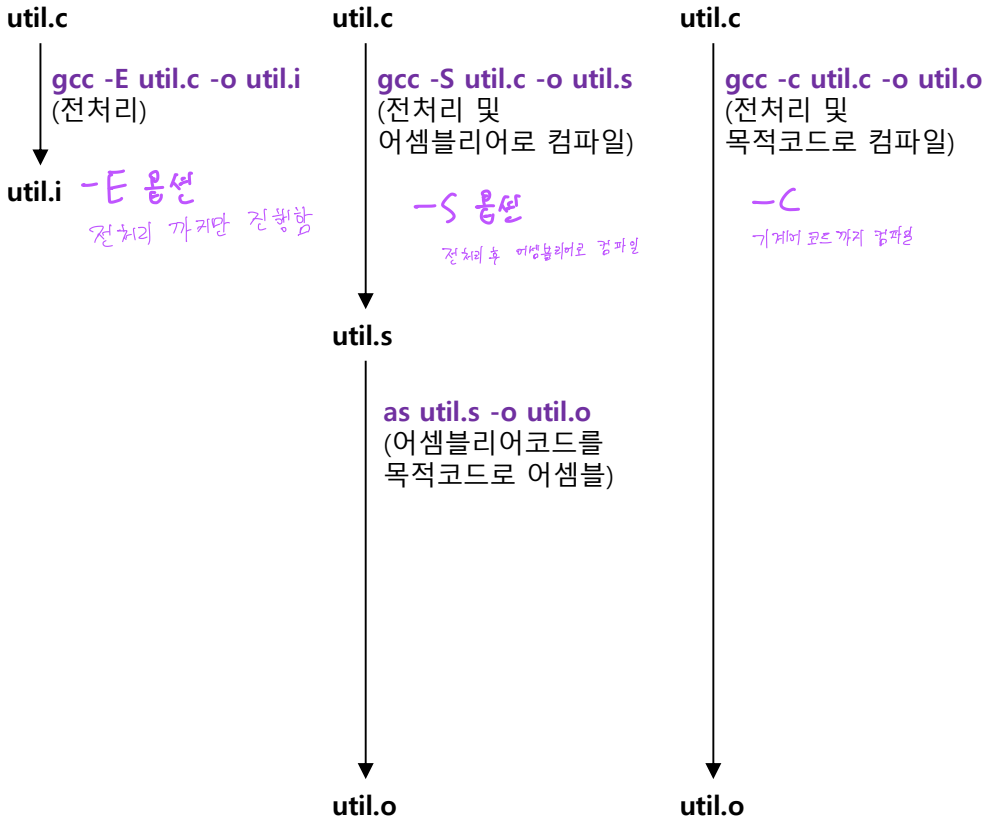
```
int ft(int x){
    return 100 +x;
}
```

util.s

```
...
push    ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
mov     ebp, esp
.cfi_def_cfa_register 5
mov     eax, DWORD PTR [ebp+8]
add     eax, 100
pop     ebp
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
...
```

util.o

```
...
0:   55          push   ebp
1:   89 e5       mov    ebp,esp
3:   8b 45 08    mov    eax,DWORD PTR [ebp+0x8]
6:   83 c0 64    add    eax,0x64
9:   5d          pop    ebp
a:   c3          ret
b:   90          nop
...
```



Compile, assemble

Reference: https://en.wikibooks.org/wiki/X86_Assembly/GNU_assembly_syntax

sum.c (C 언어)

```
int sum(int x, int y){
    return x+y;
}
```

(전처리 및 어셈블리어로 **컴파일**)

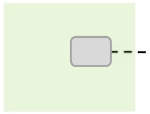
```
gcc -S sum.c -o sum.s
```

sum.s (어셈블리어)

```
...
push    ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
mov     ebp, esp
.cfi_def_cfa_register 5
mov     edx, DWORD PTR [ebp+8]
mov     eax, DWORD PTR [ebp+12]
add     eax, edx
pop     ebp
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
...
```

(**어셈블**) `as sum.s -o sum.o`

sum.o (목적파일)



```
...
0:   55          push ebp
1:   89 e5       mov  ebp,esp
3:   8b 55 08    mov  edx,DWORD PTR [ebp+0x8]
6:   8b 45 0c    mov  eax,DWORD PTR [ebp+0xc]
9:   01 d0       add  eax,edx
b:   5d          pop  ebp
c:   c3          ret
...
```

(역어셈블) `objdump -d -Mintel sum.o`

- 목적파일의 내용 출력
- Disassembler로 이용 가능

`gcc -S sum.c -o sum.s -masm=intel`

- x86 어셈블리언어는 Intel syntax와 AT&T syntax로 작성 가능
- NASM(Netwide assembler) 등 많은 x86 어셈블러들은 Intel 스타일을 따르며, GAS(GNU Assembler)는 디폴트로 AT&T 스타일을 따름

Disassembler
역어셈블러 프로그램

sum.c (C 언어)

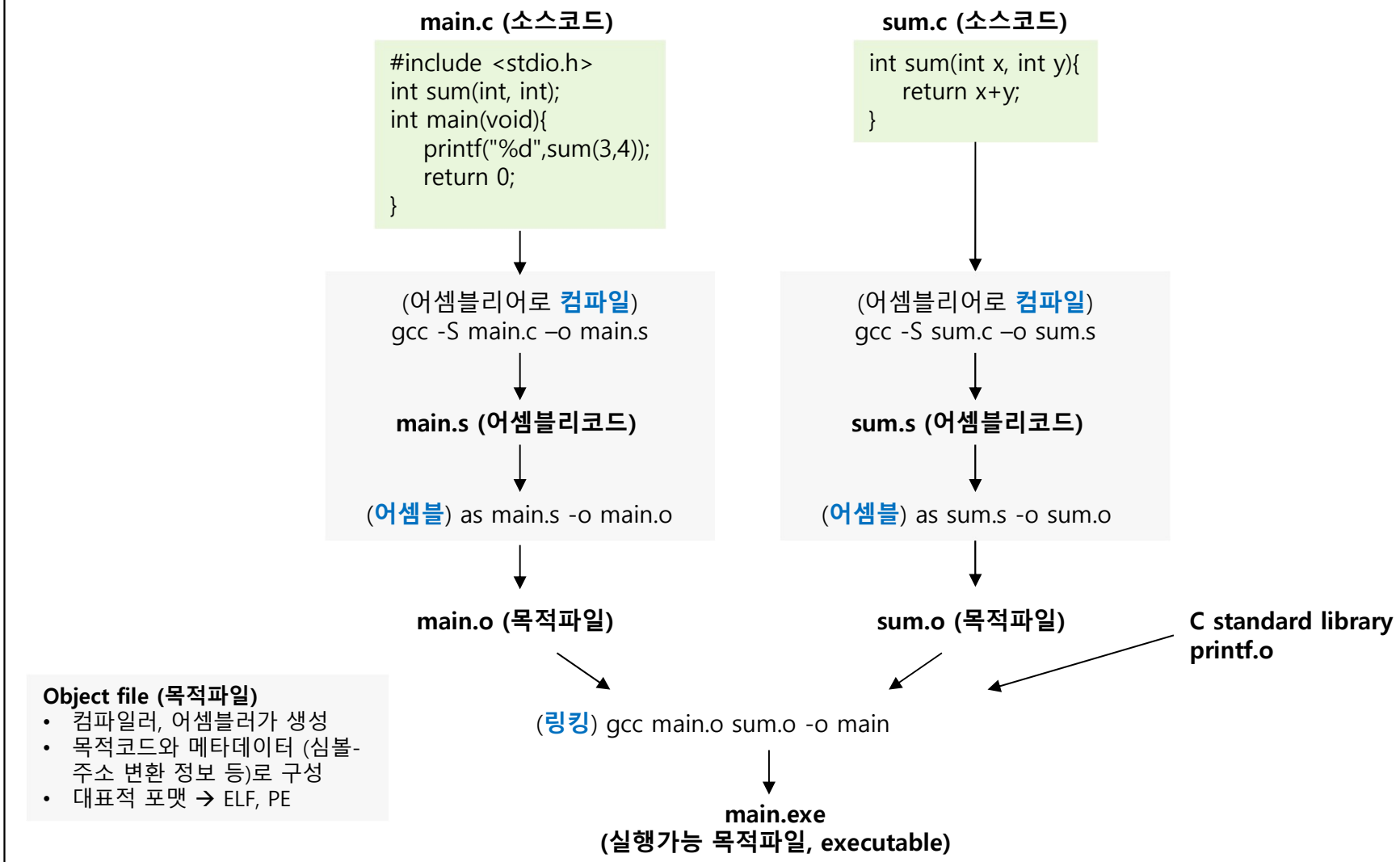
```
int sum(int x, int y){
    return x+y;
}
```

(전처리 및 목적코드로 **컴파일**)

```
gcc -c sum.c -o sum.o
```

sum.o (목적파일)

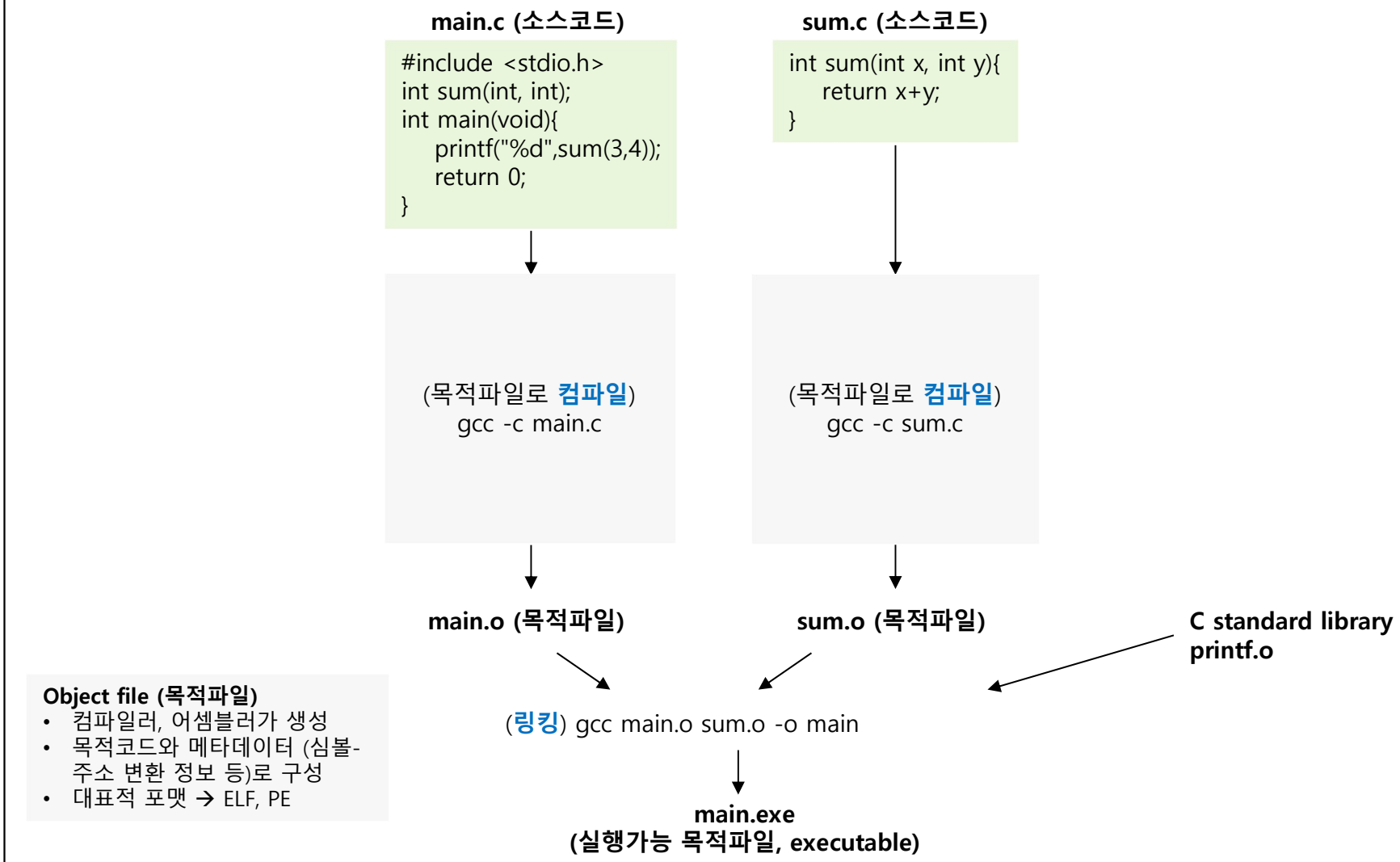
Compile, assemble, link (1/2)



Object file (목적파일)

- 컴파일러, 어셈블러가 생성
- 목적코드와 메타데이터 (심볼-주소 변환 정보 등)로 구성
- 대표적 포맷 → ELF, PE

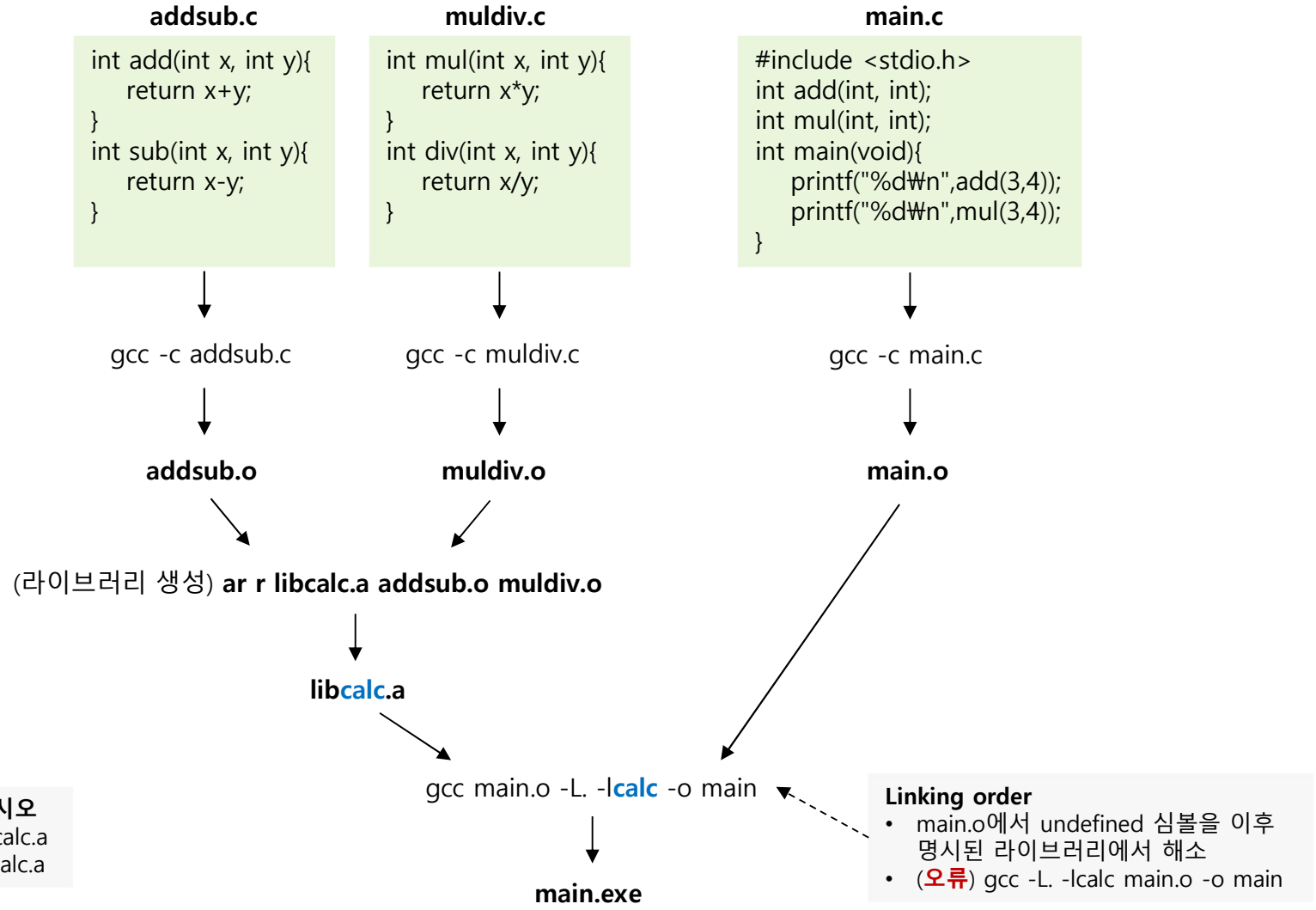
Compile, assemble, link (2/2)



Object file (목적파일)

- 컴파일러, 어셈블러가 생성
- 목적코드와 메타데이터 (심볼-주소 변환 정보 등)로 구성
- 대표적 포맷 → ELF, PE

Library & linking

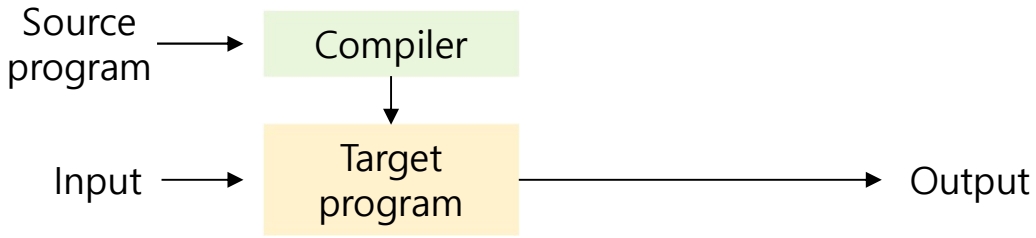


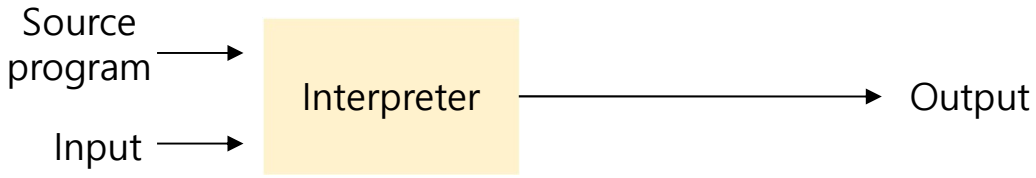
Standard library

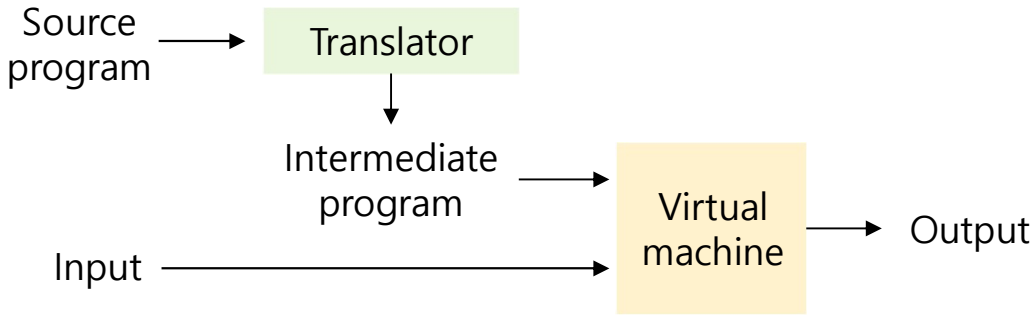
Standard library
- 한 프로그래밍 언어에서, 프로그램 작성 시 공통으로 사용되는 요소들(함수 등)의 모음
- 표준라이브러리에는 입출력 함수, 문자열 처리 함수, 수학 함수/상수, 자료구조/알고리즘 구현체 등이 포함될 수 있으며 포함 내용은 언어마다 다름

C standard library
- printf(), scanf(), strlen(), strcmp(), malloc(), free(), sqrt(), sin(), qsort(), …

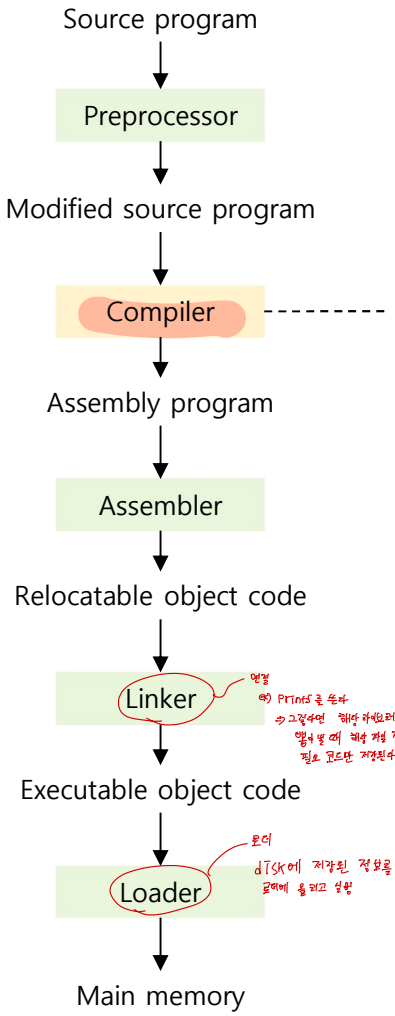
Compiler, interpreter, hybrid

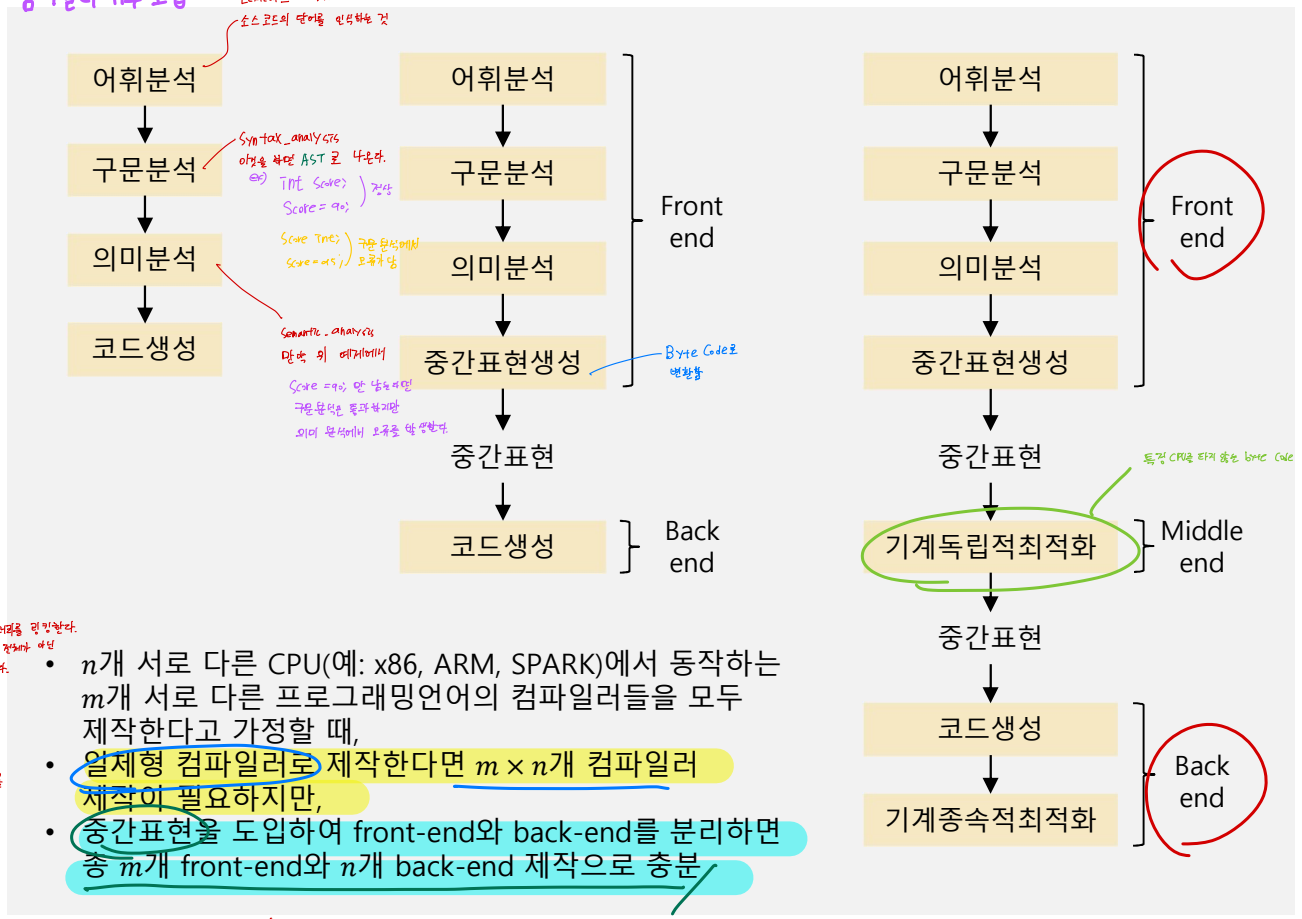




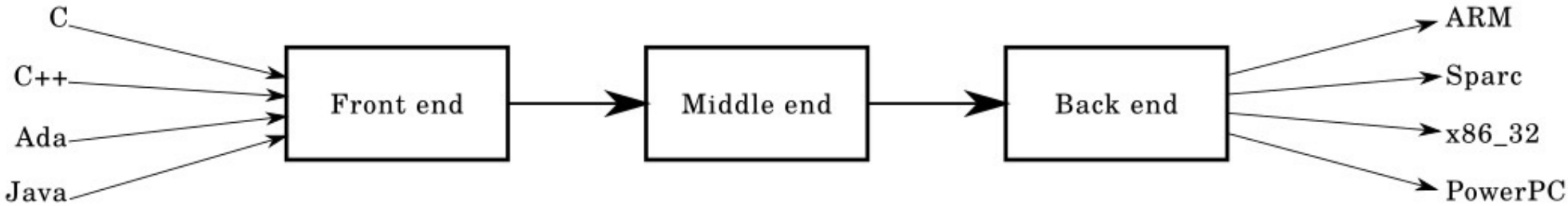


Compiler 내부 단계 (1/2)





- n개 서로 다른 CPU(예: x86, ARM, SPARK)에서 동작하는 m개 서로 다른 프로그래밍언어의 컴파일러들을 모두 제작한다고 가정할 때,
- 일체형 컴파일러로 제작한다면 $m \times n$개 컴파일러 제작이 필요하지만,
- 중간표현을 도입하여 front-end와 back-end를 분리하면 총 m개 front-end와 n개 back-end 제작으로 충분



과거에는 일체형 컴파일러가 많이 사용되었다.

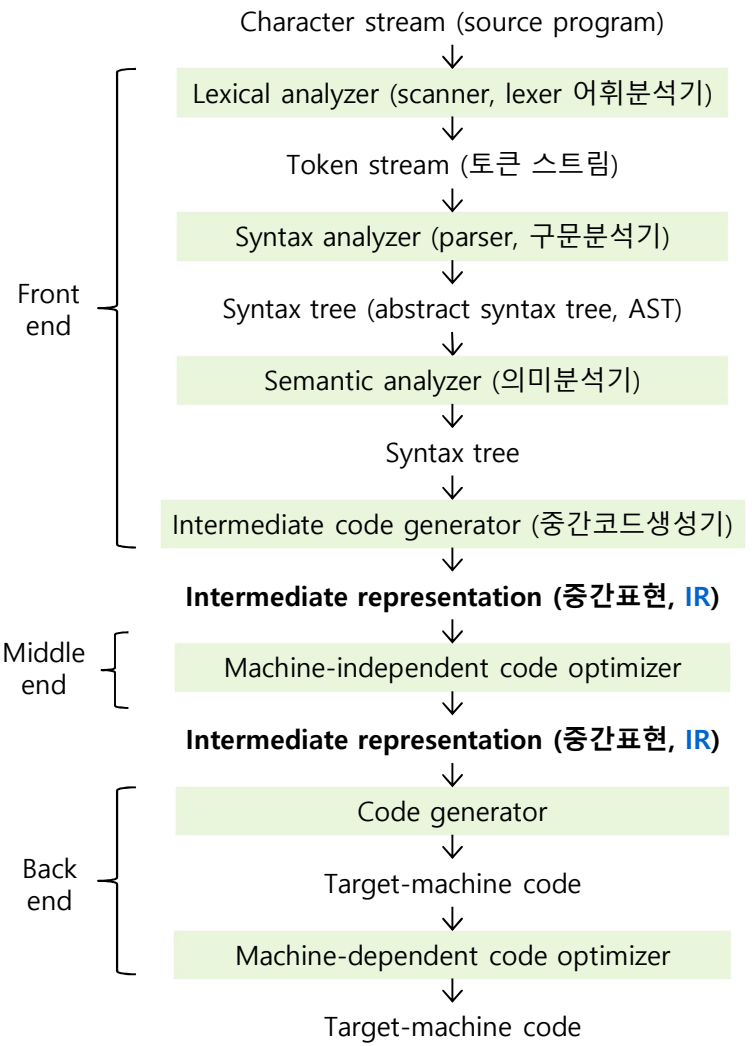
CPU 종류가 늘어나며 (ARM, x86, SPARC, PowerPC) 등등

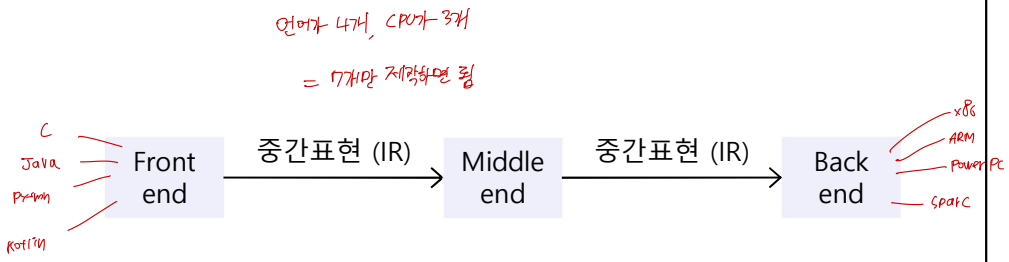
CPU당 컴파일러를 여러개 제작하게 됨

그래서 Front-end, Middle-end, back-end 를 만들어

프론트, 백엔드만 바꾸어 컴파일 가능하게 함

Compiler 내부 단계 (2/2)

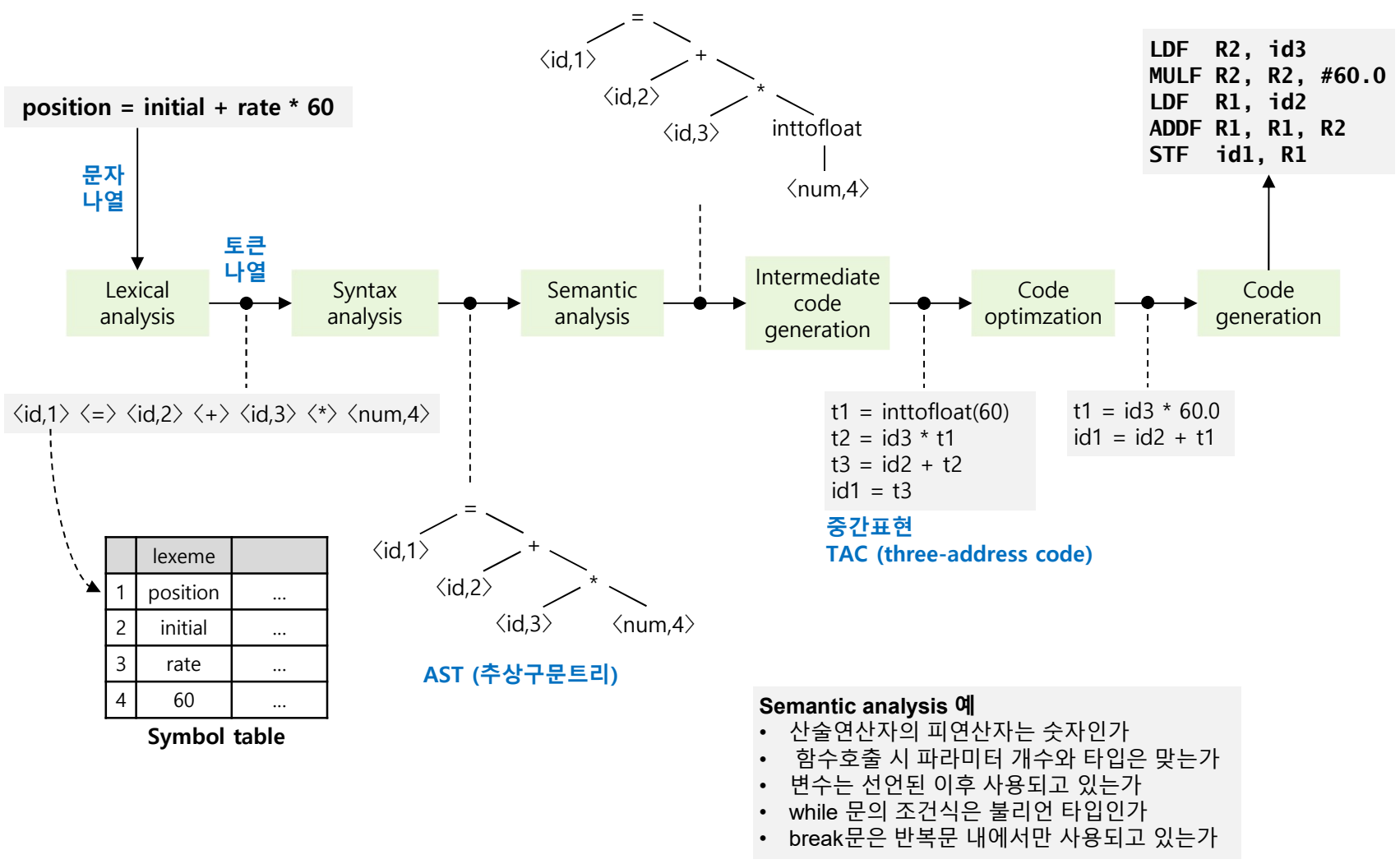




Front end와 back end를 IR로 분리

- **Front end**는 소스프로그램을 입력으로 받아, 어휘분석, 구문분석, 의미분석을 수행한 후 (소스프로그램에 대한) 중간표현(Intermediate Representation, **IR**)을 생성
- **Middle end**는 중간표현에 기계(CPU) 독립적 최적화 수행.
  중간표현의 예로 AST(abstract syntax tree)와 TAC(three-address code)가 있음
- **Back end**는 (소스프로그램에 대한) 중간표현을 입력으로 받아, 기계 종속적 최적화 수행 후, 목표 기계를 위한 어셈블리 코드나 기계어 코드를 생성
- GNU Compiler Collection, Clang 등에서 이러한 접근법 채택
- n개 서로 다른 CPU(예: x86, ARM, SPARK)에서 동작하는 m개 서로 다른 프로그래밍언어의 컴파일러들을 모두 제작하기 위해, 총 m개 front-end와 n개 back-end만 필요하지만, front-end/back-end 분리 없이 일체형 컴파일러로 제작한다면
  - $m \times n$개 컴파일러 제작 필요
  - 새로운 CPU가 등장한 경우 m개 컴파일러 추가 제작 필요
  - 새로운 언어가 등장한 경우 n개 컴파일러 추가 제작 필요

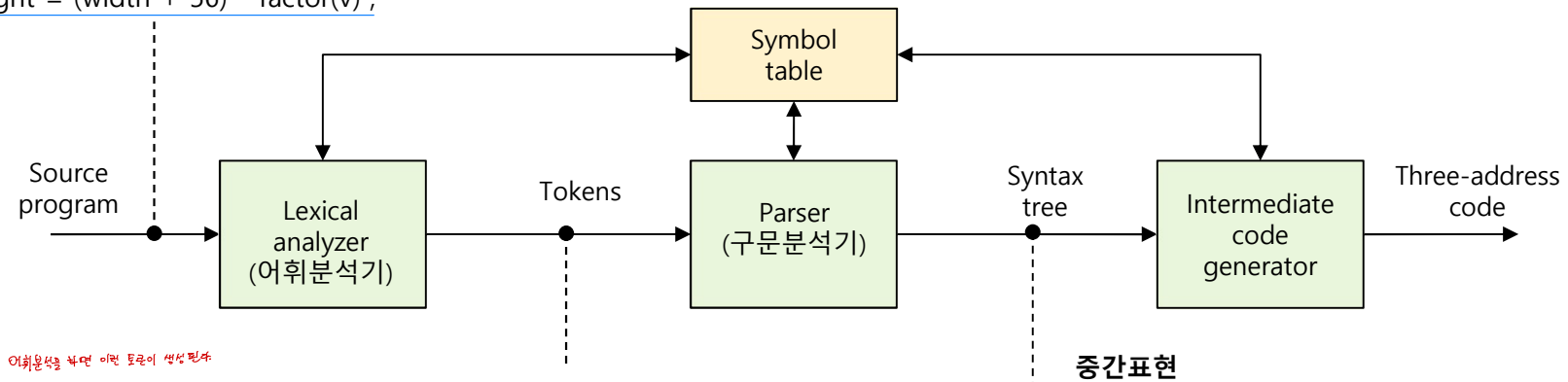
Compiler 내부 단계 (2/2)



	lexeme	
1	position	...
2	initial	...
3	rate	...
4	60	...

Compiler 내부 단계 일부 (1/3)

height = (width + 56) * factor(v) ;



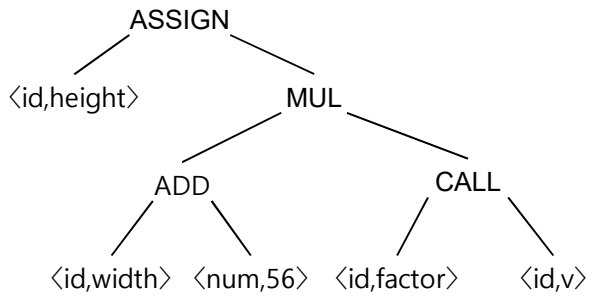
어휘분석을 하면 이런 토큰이 생성된다

〈id,height〉 〈=〉 〈(〉 〈id,width〉 〈+〉 〈num,56〉 〈)〉 〈*〉 〈id,factor〉 〈(〉 〈id,v〉 〈)〉 〈;〉

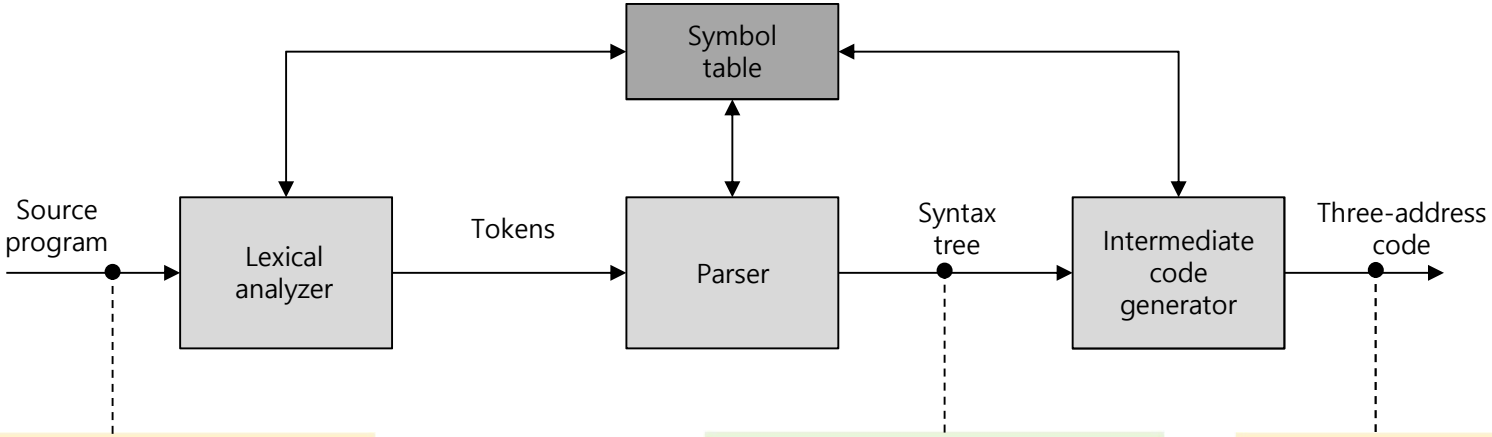
- factor가 식별자인 것은 알지만 함수 이름인지 변수 이름인지는 알지 못함
- 이 문장만으로 width의 타입을 알지 못함
- +는 정수 덧셈일 수도 있고, 실수 덧셈일 수도 있고, 문자열 연결일 수도 있음

중간표현

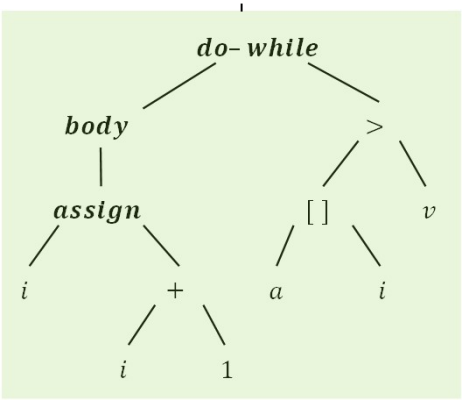
- 컴파일러의 front end는 소스프로그램에 대한 중간표현(intermediate representation)을 생성하고, back end는 중간표현으로부터 목적프로그램을 생성
- 두 가지 대표적 중간표현으로 abstract syntax tree와 three-address code가 있음



Compiler 내부 단계 일부 (2/3)




```
do i=i+1; while ( a[i] < v);
```

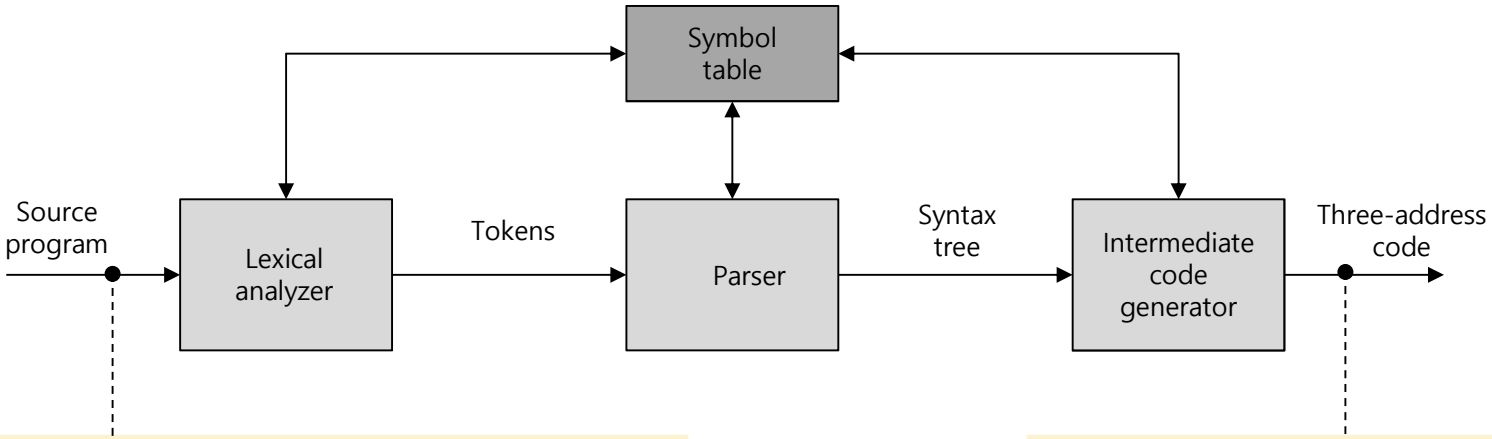



```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

↳ 3번에 참일경우 1으로 계속실행

Program -> Block
Block -> **{** Decls Stmts **}**
Decls -> Decls Decl | ϵ
Decl -> Type **id** ;
Type -> Type **[num]** | **int** | **char** | **bool** | **float**
Stmts -> Stmts Stmt | ϵ
Stmt -> Loc **=** Bool **;** | **break ;** | Block
Stmt -> **if (** Bool **)** Stmt | **if (** Bool **)** Stmt **else** Stmt
Stmt -> **while (** Bool **)** Stmt | **do** Stmt **while (** Bool **)** ;
Loc -> Loc **[** Bool **]** | **id**
Bool -> Bool **||** Join | Join
Join -> Join **&&** Equality | Equality
Equality -> Equality **==** Rel | Equality **!=** Rel | Rel
Rel -> Expr **<** Expr | Expr **<=** Expr | Expr **>=** Expr | Expr **>** Expr | Expr
Expr -> Expr **+** Term | Expr **-** Term | Term
Term -> Term ***** Unary | Term **/** Unary | Unary
Unary -> **!** Unary | **-** Unary | Factor
Factor -> **(** Bool **)** | Loc | **Num** | **Real** | **true** | **false**

Compiler 내부 단계 일부 (3/3)




```
{
   int i; int j;
   float[100] a; float v; float x;
   while ( true ) {
      do i = i+1; while ( a[i] < v );
      do j = j-1; while ( a[j] > v );
      if ( i >= j ) break;
      x = a[i]; a[i] = a[j]; a[j] = x;
   }
}
```

```
 1: i = i + 1
 2: t1 = a [ i ]
 3: if t1 < v goto 1
 4: j = j - 1
 5: t2 = a [ j ]
 6: if t2 > v goto 4
 7: ifFalse i >= j goto 9
 8: goto 14
 9: x = a [ i ]
10: t3 = a [ j ]
11: a [ i ] = t3
12: a [ j ] = x
13: goto 1
14:
```

References

- 박두순. (2016). (내공 있는 프로그래머로 길러주는)컴파일러의 이해. 한빛아카데미.
- 창병모. (2021). 프로그래밍 언어론 : 원리와 실제. 인피니티북스.
- Aho, A., Lam, M., Sethi, R., Ullman, J. (2006). Compilers: Principles, Techniques, and Tools (2nd Ed.). Addison Wesley.
- Nystrom, R. Crafting interpreter. https://craftinginterpreters.com/
- Sebesta, R. (2012). Concepts of Programming Languages (10th. ed.). Pearson.
- Thain, D. (2023). Introduction to Compilers and Language Design.
- Paxson, V. (1995). Flex, version 2.5.
- Donnelly, C., Stallman, R. (2008). Bison.
- LexAndYacc.pdf (epaperpress.com)
- Tom Niemann, LEX & YACC. http://epaperpress.com/lexandyacc