Compiler, Assembler, and Linker

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University

msryu@hanyang.ac.kr

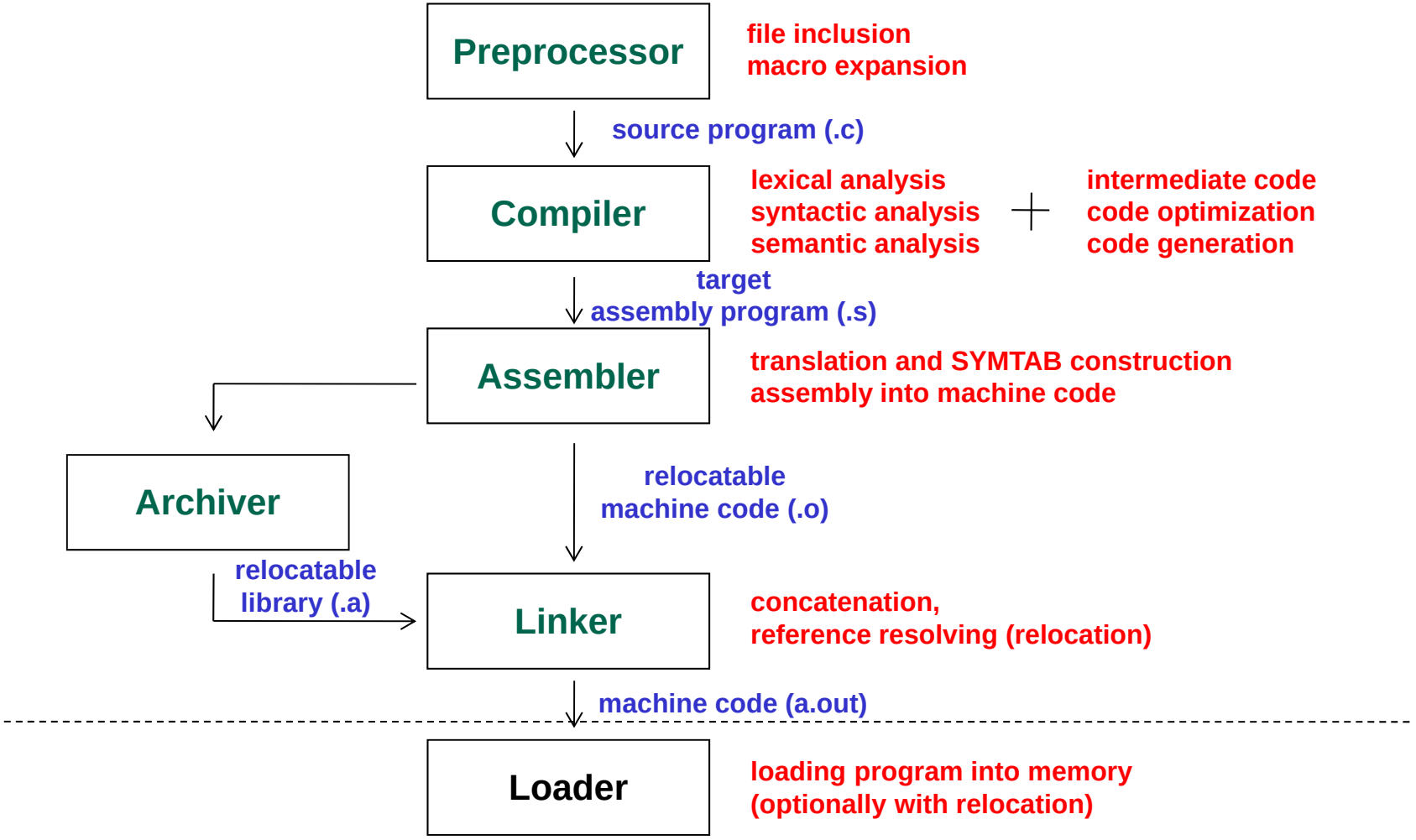
Contents

- What is a Compilation?
- Preprocessor
- Compiler
- Assembler
- Linker
- Loader

What is a Compilation?

- A compilation is a translation process that reads a program written in one language--the source language--and translate it into an equivalent program in another language--the target language

The Complete Process



Contents

- What is a Compilation?
- Preprocessor
- Compiler
- Assembler
- Linker
- Loader

Preprocessor

- **Inclusion**
  - Copies the full content of a included file into the current file at the point at which the directive occurs
  - Eg.) #include "…" directive in 'C'

    ODR (one definition rule) - 변수 정의 한번만
- **Macros**
  - Replace each macro call, in-line, with the corresponding macro definition
  - E.g.) #define … … directive in 'C'
- **Conditional compilation (include or macro guard)**
  - In combination with macros, remove code fragments that need not be compiled
  - E.g) #ifdef, #ifndef, #endif directives in 'C'
- **Other compiler directives**
  - #pragma once (to ensure single inclusion)

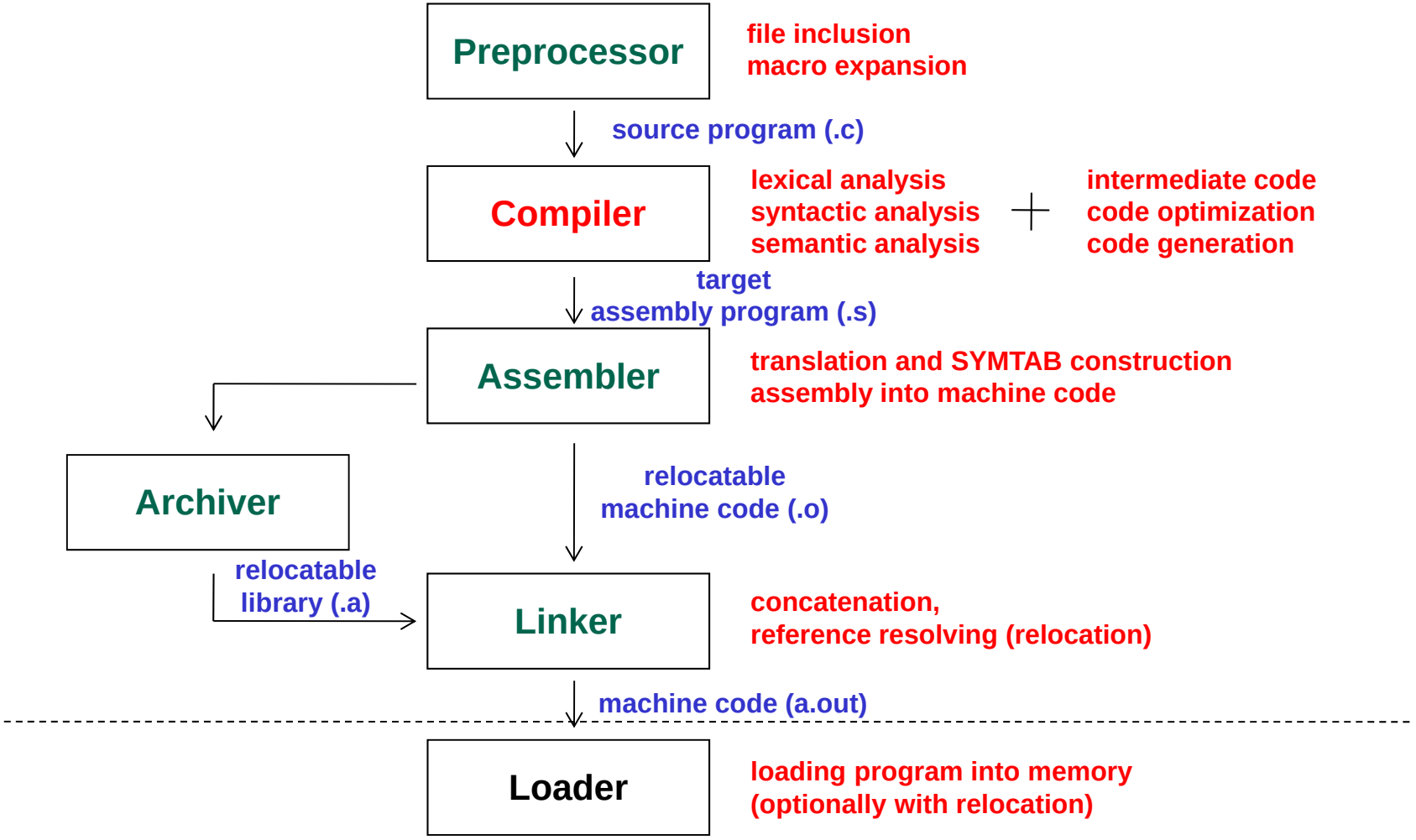
Notes on #pragma

- The #pragma directives offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C and C++ languages
  - Pragmas are machine- or operating system-specific by definition, and are usually different for every compiler
  - Each implementation of C and C++ supports some features unique to its host machine or operating system
  - Some programs, for instance, need to exercise precise control over the memory areas where data is placed or to control the way certain functions receive parameters

Contents

- What is a Compilation?
- Preprocessor
- Compiler
- Assembler
- Linker
- Loader

The Complete Process



Compiler

- Analysis
  - Lexical analysis
  - Syntactic analysis
  - Semantic analysis
- Synthesis
  - Intermediate code generation
  - Code optimization
  - Code generation

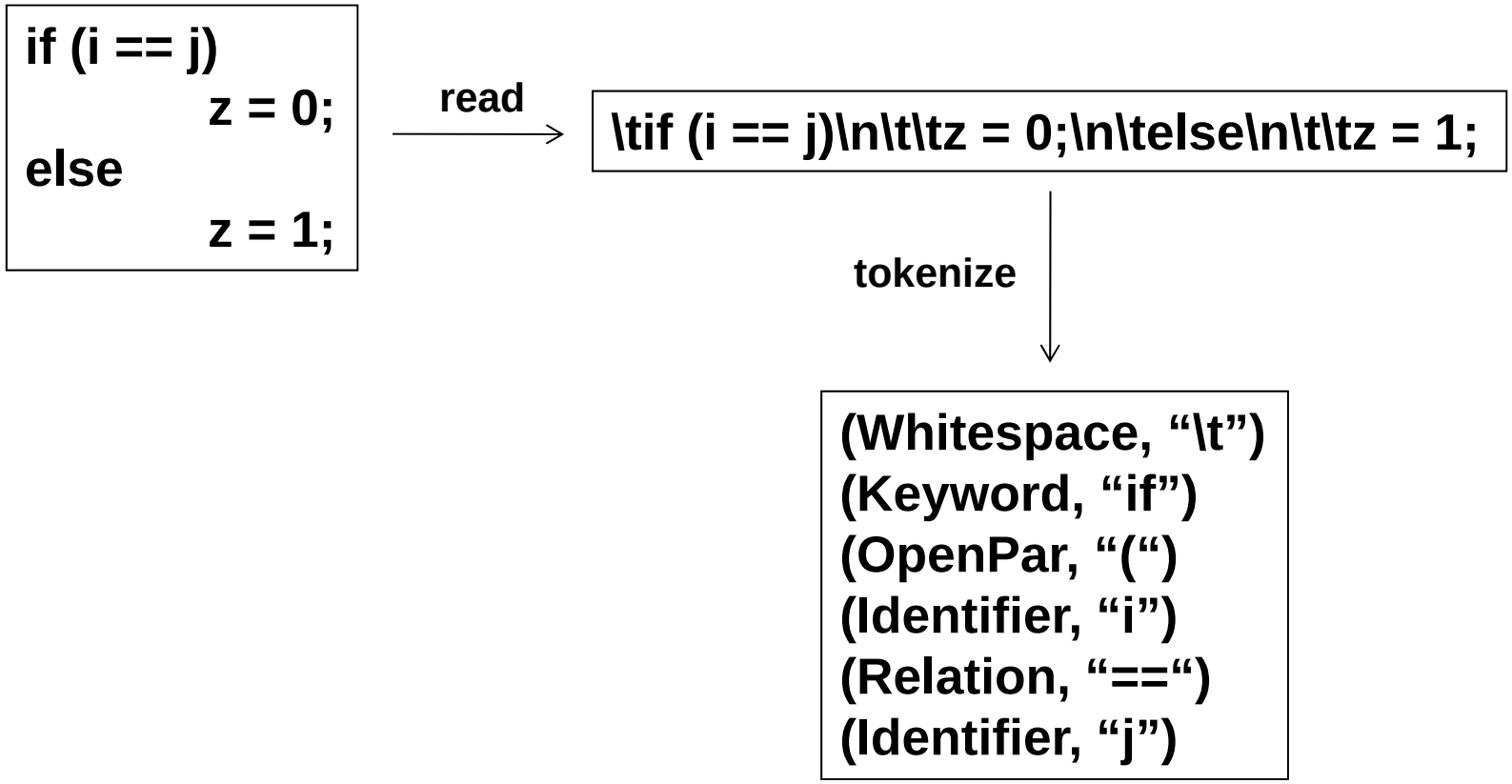
Lexical Analysis

- Lexical analysis is also known as linear addressing or scanning

- Lexical analysis;
  - Reads the source program from left-to-right
  - Identifies the lexemes in the input list of characters and categorize them into tokens
    - Lexeme (어휘) and lexicon (어휘 집합)

- A token is a syntactic category
  - Token = lexeme's type (+ its value)
  - In English: noun, verb, adjective, …
  - In a programming language: Identifier, Integer, Keyword, …

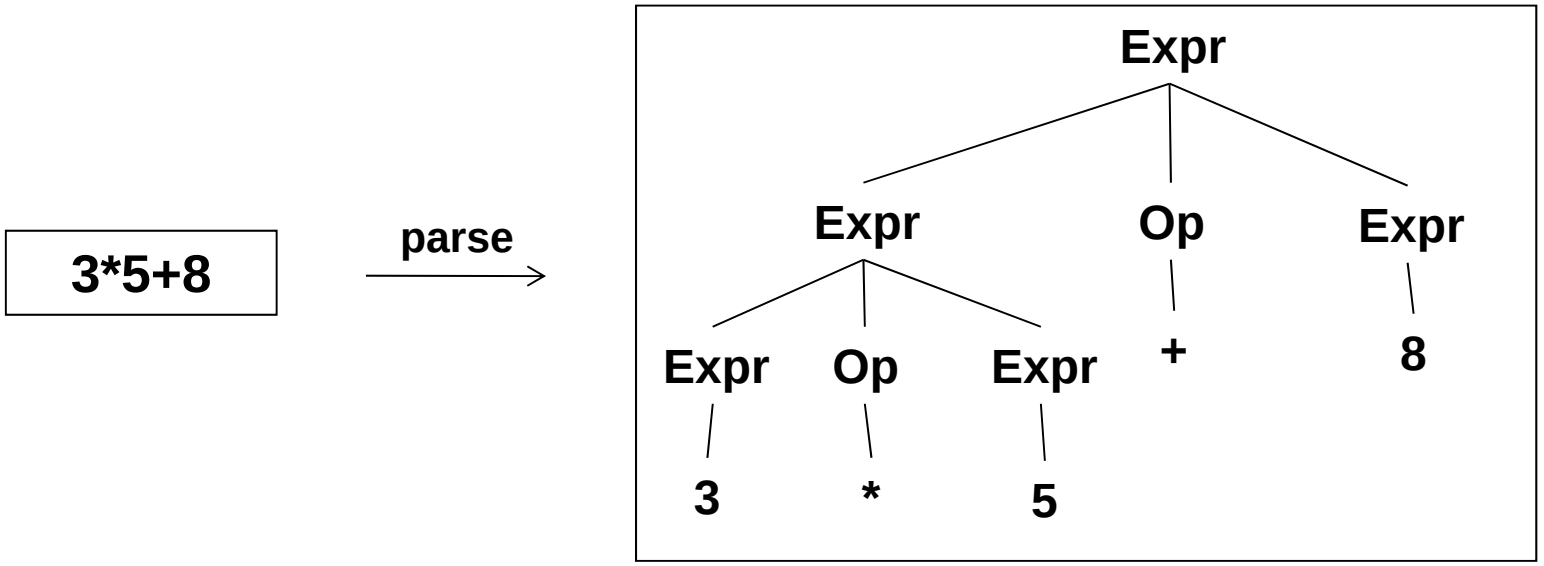
Lexical Analysis



Syntactic Analysis

- Syntactic analysis is also known as parsing
- Syntactic analysis;
  - Analyzes a sequence of tokens to determine grammatical structure with respect to a given formal grammar
  - Checks for syntax errors at the same time
- If there exists no syntax error, the result is often represented by a parse tree

Syntactic Analysis



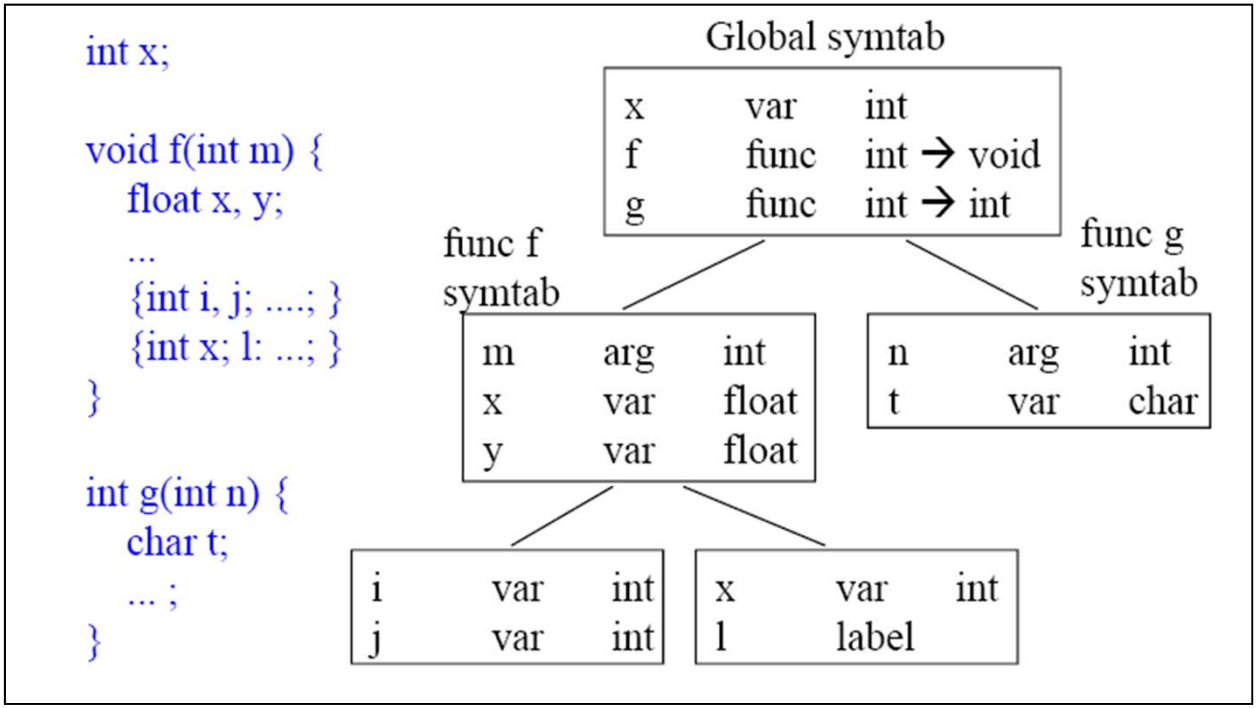
Semantic Analysis

- **Semantic analysis performs;**
  - **Semantic checks for;**
    - Types (checking for type errors)
    - Scopes (variables declared before use, ODR violations)
  - **Construct symbol tables**
    - Variable names and function names + information
  - **Others**
    - Object binding (associating variable and function references with their definitions)
    - Definite assignment (requiring all local variables to be initialized before use)
    - Rejecting incorrect programs or issuing warnings

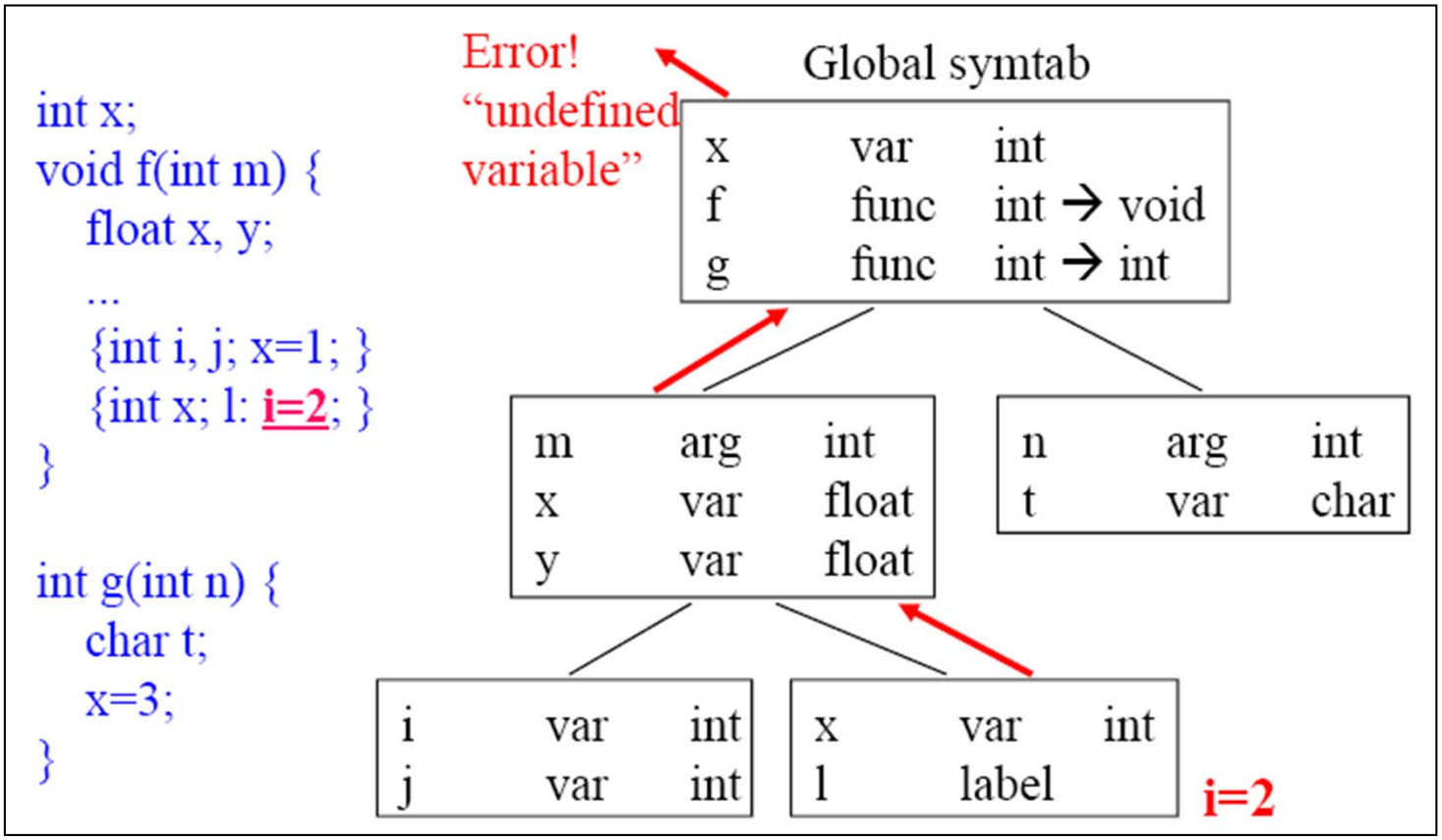
- **Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase**

Semantic Analysis (Symbol Tables)

NAME	KIND	TYPE	ATTRIBUTES
foo	func	int,int → int	extern
m	arg	int	
n	arg	int	const
tmp	var	char	const



Semantic Analysis (Scope Checking)



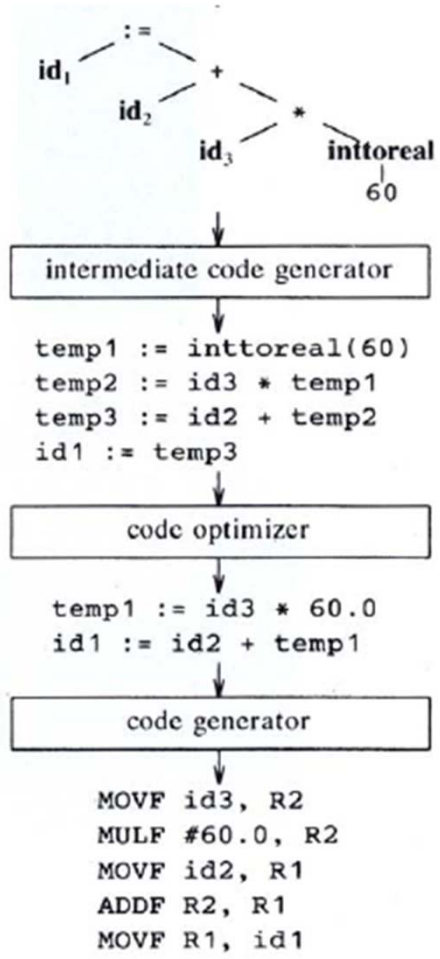
Intermediate Code Generation and Code Optimization

❒ Intermediate code generation

- Generates an explicit inter-mediate representation of the source program
- Often emulates a virtual (RISC) processor that provides a simple instruction set architecture

❒ Code optimization

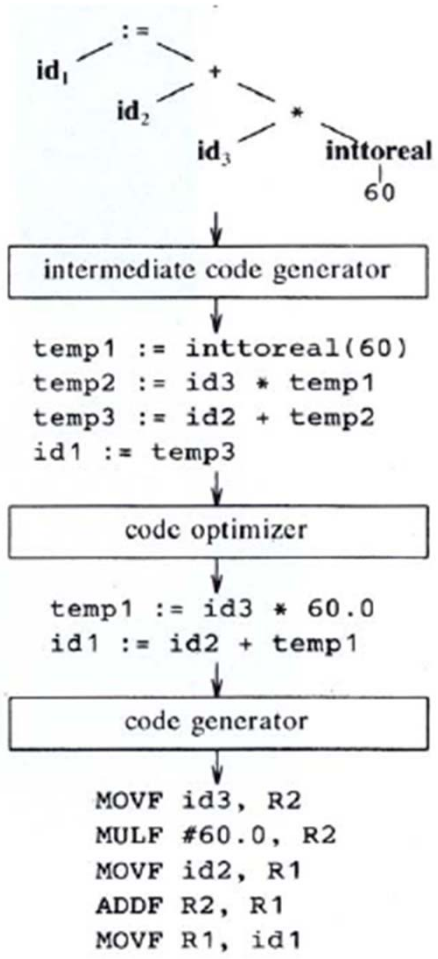
- Attempts to improve the intermediate code, so that faster-running machine code will result



Code Generation

❒ Code generation

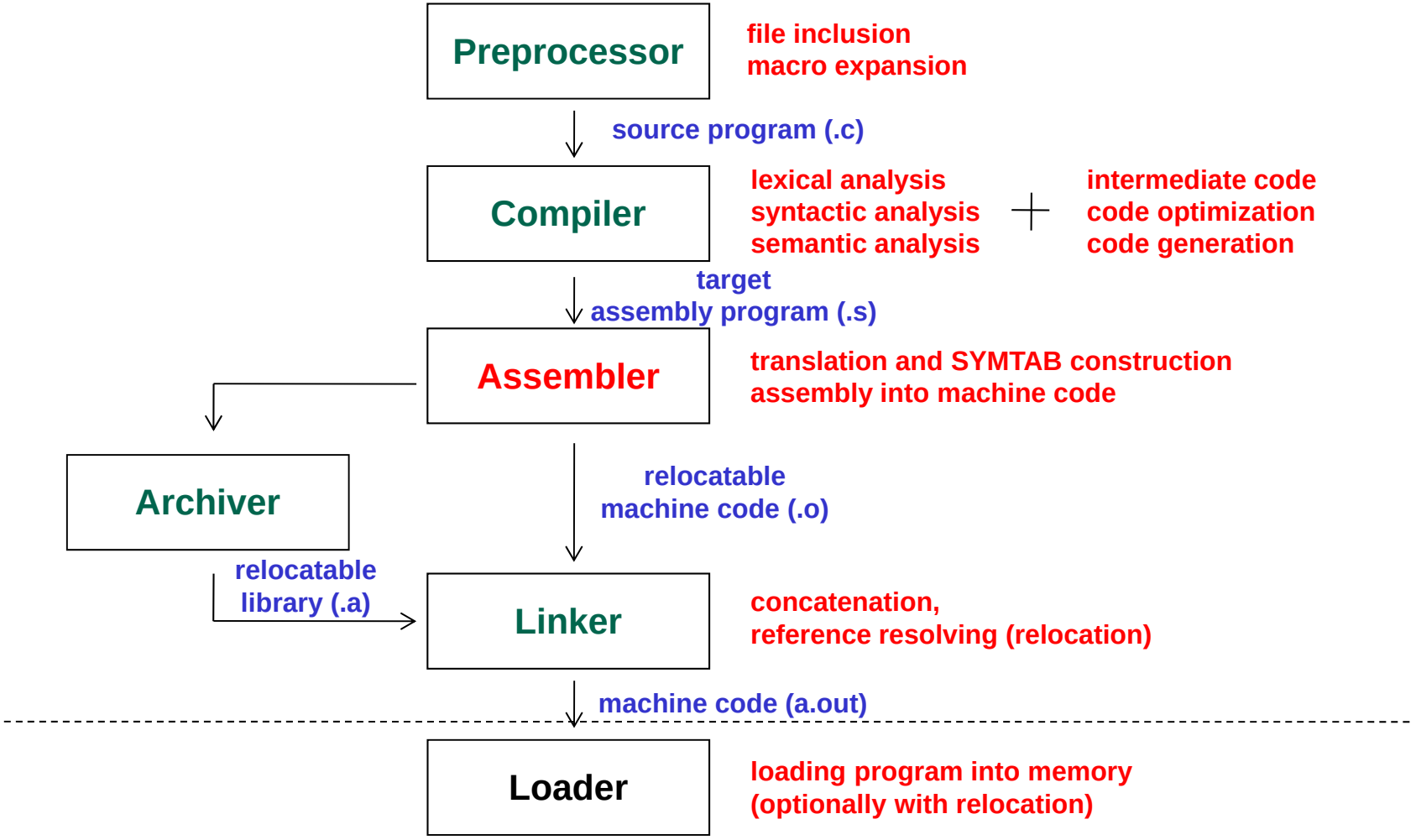
- Generates target code, consisting normally of relocatable machine code or assembly code
- Memory locations are selected for each of the variables used by the program
- Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task
- A crucial aspect is the assignment of variables to registers



Contents

- What is a Compilation?
- Preprocessor
- Compiler
- Assembler
- Linker
- Loader

The Complete Process



Assembler

- Assembler;
  - Converts mnemonic operation codes to their machine language codes
  - Converts symbolic (e.g., jump labels, variable names) operands to their machine addresses
  - Uses proper addressing modes and formats to build efficient machine instructions
  - Translates data constants into internal machine representations
  - Outputs the object program and provide other information (e.g., for linker and loader)

Assembler

instruction location counter

Label	Opcode	Operands	Comments	Length	ILC
MARIA:	MOV	EAX, I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
ROBERTA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I * I	2	117
	IMUL	EBX, EBX	EBX = J * J	3	119
	IMUL	ECX, ECX	ECX = K * K	3	122
MARILYN:	ADD	EAX, EBX	EAX = I * I + J * J	2	125
	ADD	EAX, ECX	EAX = I * I + J * J + K * K	2	127
STEPHANY:	JMP	DONE	branch to DONE	5	129

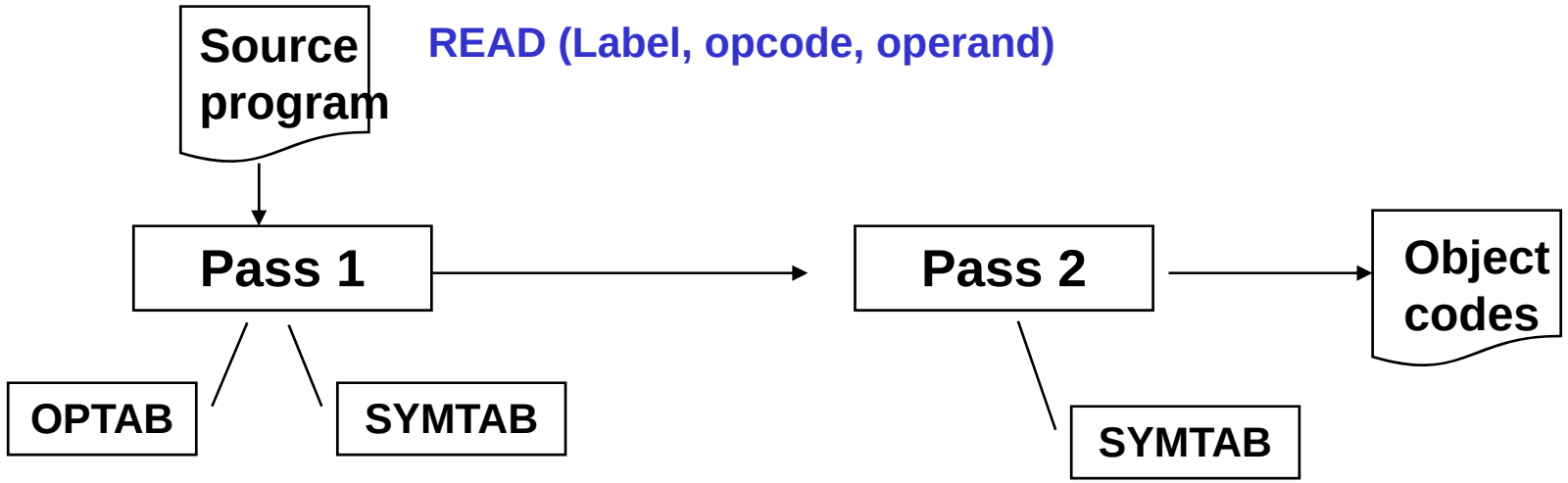
Assembler

Line	ILC	Source Statement			Machine Code
5	1000	COPY	START	1000	
10	1000	FIRST	STL	RETADR	141033
15	1003	CLOOP	JSUB	RDREC	482039
20	1006		LDA	LENGTH	001036
25	1009		COMP	ZERO	281030
30	100C		JEQ	ENDFIL	301015
35	100F		JSUB	WRREC	482061
40	1012		J	CLOOP	3C1003
45	1015	ENDFIL	LDA	EOF	00102A
50	1018		STA	BUFFER	0C1039
55	101B		LDA	THREE	00102D
60	101E		STA	LENGTH	0C1036
65	1021		JSUB	WRREC	482061
70	1024		LDL	RETADR	081033
75	1027		RSUB		4C0000
80	102A	EOF	BYTE	C'EOF'	454F46
85	102D	THREE	WORD	3	000003
90	1030	ZERO	WORD	0	000000
95	1033	RETADR	RESW	1	
100	1036	LENGTH	RESW	1	
105	1039	BUFFER	RESB	4096	
110		.			

Two Pass Assembler

- Pass 1
  - Assign addresses to all statements in the program
  - Save the values (addresses) assigned to all labels (including label and variable names) for use in Pass 2 (deal with forward references)
  - Perform some processing of assembler directives that can affect address assignment
- Pass 2
  - Assemble instructions (generate opcode and look up addresses)
  - Perform processing of assembler directives not done in Pass 1
  - Write the object program and the assembly listing

Two Pass Assembler



Operation Code Table (OPTAB)

- Content
  - The mapping between mnemonic and machine code
  - Also include the instruction format, available addressing modes, and length information
- Characteristic
  - Static table (the content will never change)
- Implementation
  - Array or hash table because the content will never change, we can optimize its search speed
- In pass 1, OPTAB is used to look up and validate mnemonics in the source program
- In pass 2, OPTAB is used to translate mnemonics to machine instructions

Instruction Location Counter (ILC)

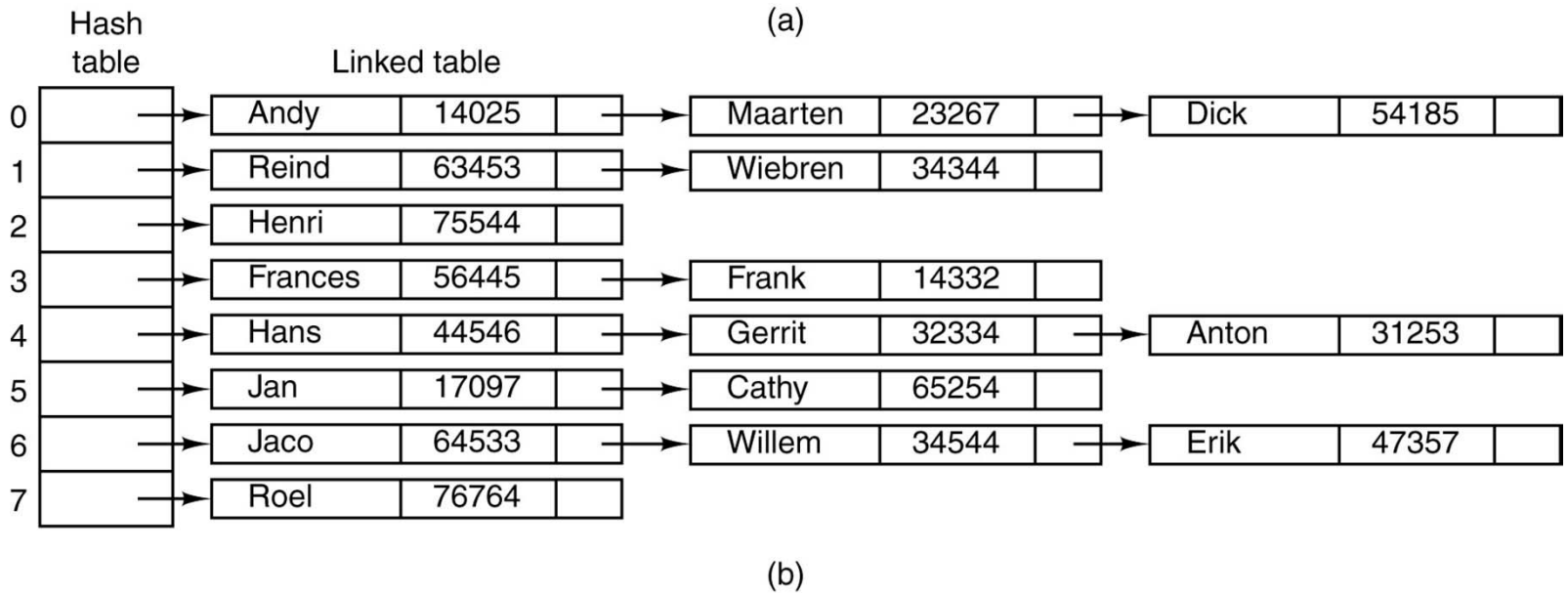
- This variable can help in the assignment of addresses
- It is initialized to the beginning address specified in the START statement
- After each source statement is processed, the length of the assembled instruction and data area to be generated is added to LOCCTR
- Thus, when we reach a label in the source program, the current value of LOCCTR gives the address to be associated with that label

Symbol Table (SYMTAB)

- **Content**
  - **Include the label name and value (address) for each label in the source program**
  - **Include type and length information**
  - **With flag to indicate errors (e.g., a symbol defined in two places)**
- **Characteristic**
  - **Dynamic table (symbols may be inserted, deleted, or searched in the table)**
- **Implementation**
  - **Hash table can be used to speed up search**
  - **Because variable names may be very similar (e.g., LOOP1, LOOP2), the selected hash function must perform well with such non-random keys**

Symbol Table (SYMTAB)

❒ Hash coding of symbol tables



Object Code

- **An object code in the end contains the following information:**
  - **A header that says where in the files the sections below are locates**
  - **A (concatenated) text segment, which contains all the source code (with some missing addresses)**
  - **A (concatenated) data segment (which may combine the data and the bss segments)**
  - **Relocation Table: identifies lines of code that need to be "fixed"** absoulte addressing - 컴파일 시간때 정해진 physical addressing에 memory loading
    relative addressing - ex) pc에 relative한 addressing에 memory loading -> os의 자유도 증가
  - **Symbol Table: list of this file's referenceable labels**
    - Functions and global variables
  - **Perhaps debugging information (is compiled with -g from a high-level programming language)**
  - **Source code line numbers, etc.**

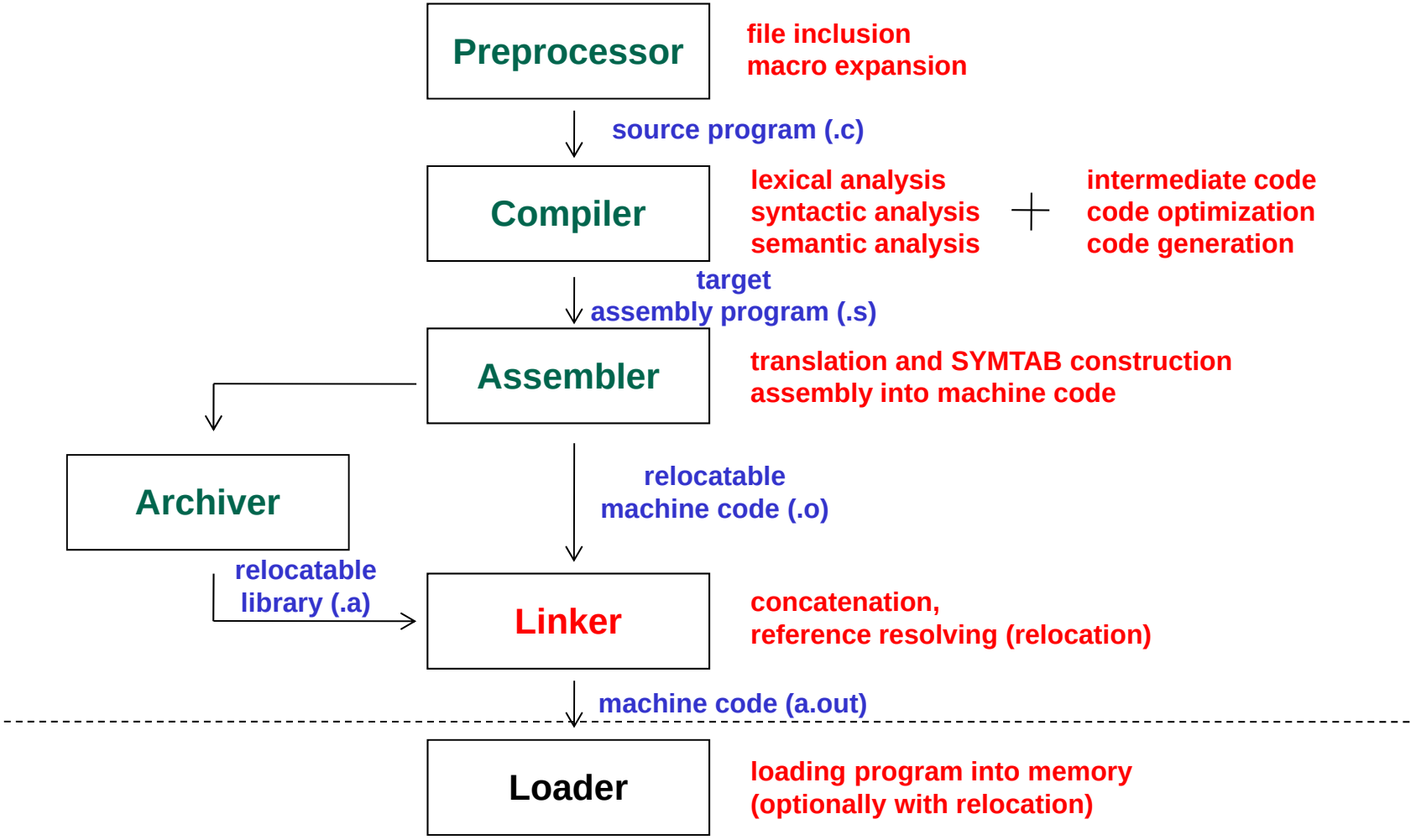
Object File Format

- **Object files are often called "binaries“**
- **It is common for the same file format to be used both as linker input and output, and thus as the library and executable file format**
- **There are many different object file formats;**
  - **COFF and ELF for UNIX**
  - **COM for DOS (the simplest format)**
  - **Portable Executable (PE) for Windows**
    - For executables, object code, and DLLs, used in 32-bit and 64-bit versions of Windows operating systems
    - Used for EXE, DLL, OBJ, SYS

Contents

- What is a Compilation?
- Preprocessor
- Compiler
- Assembler
- Linker
- Loader

The Complete Process



Linker

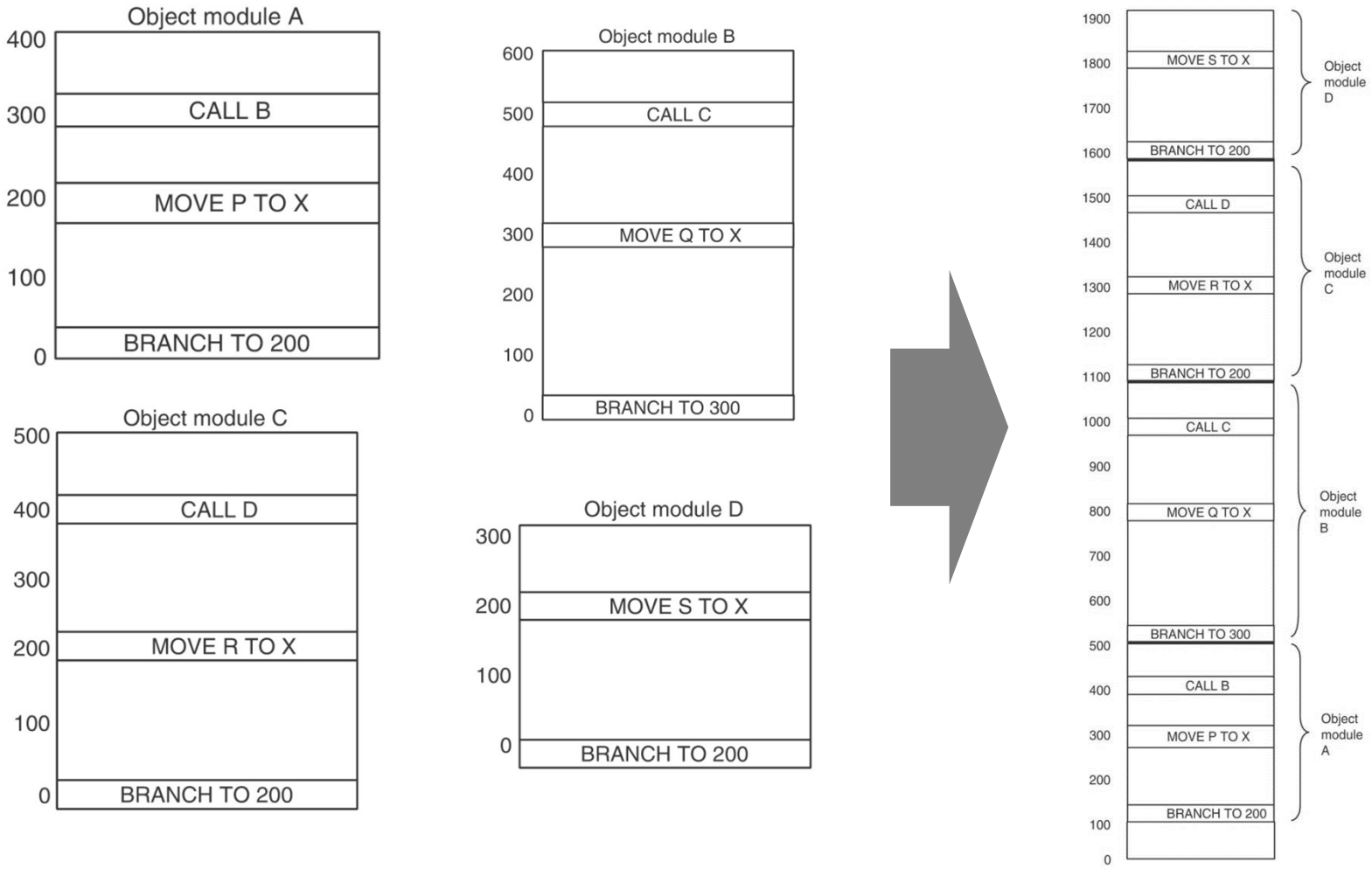
- Linker is also known as linkage editor

- Three steps
  - Step 1: concatenate all the text segments from all the .o files
  - Step 2: concatenate all the data/bss segments from all the .o files
  - Step 3: Resolve references
    - Use the relocation tables and the symbol tables to compute all absolute addresses

Linker

- The linker knows
  - The length of each text and data segment
  - The order in which they are
- Therefore the linker can compute an absolute address for each label
  - assuming the beginning of the executable file is at address 0x0
- For each label being referenced (that is for each line of code that's pointed to by the relocation table), find where it is defined
  - In the symbol table of a .o file
  - In some specified or standard library file (e.g., fprintf)
- If not found, print a "symbol not found" error message and abort
- If found in multiple tables, print a "multiply defined" error message and abort
- If found in exactly one table, replace the label by an absolute address

Linker


Object module A
400
300
CALL B
200
MOVE P TO X
100
0
BRANCH TO 200
Object module B
600
500
CALL C
400
300
MOVE Q TO X
200
100
0
BRANCH TO 300
Object module C
500
400
CALL D
300
200
MOVE R TO X
100
0
BRANCH TO 200
Object module D
300
200
MOVE S TO X
100
0
BRANCH TO 200
1900
1800
MOVE S TO X
1700
1600
BRANCH TO 200
Object module D
1500
CALL D
1400
1300
MOVE R TO X
1200
1100
BRANCH TO 200
Object module C
1000
CALL C
900
800
MOVE Q TO X
700
600
BRANCH TO 300
Object module B
500
400
CALL B
300
MOVE P TO X
200
100
BRANCH TO 200
Object module A
0

Loader

- With a linked executable, all addresses are known so that the program can run
- To actually run the program we need to use a loader, which is part of the O/S
- The loader does the following:
  - Read the executable file's header to find out the size of the text and data segments
  - Creates a new address space for the program that is large enough to hold the text and data segments, and to hold the stack (within some bounds)
  - Copies the text and data segments into the address space
  - Copies arguments passed to the program on the stack
  - Initializes the registers
  - Jump to a standard "start up routine", which sets the PC and calls the exit() system call when the program terminates