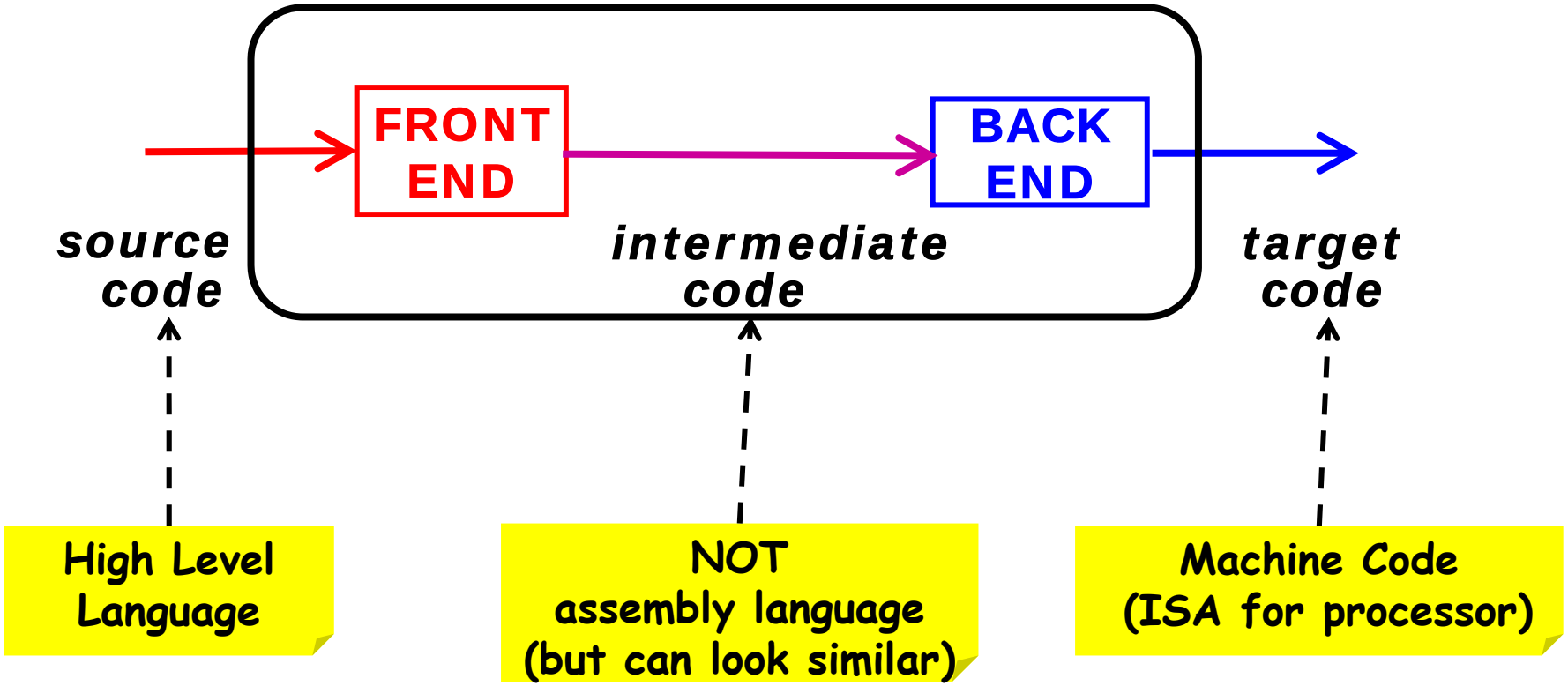
15 Compilers, Linkers, Loaders
CSC 230

Department of Computer Science
University of Victoria

M&H: chapter 6 (skipping some in-depth details)
Stallings: 15.1 to 15.4 (skipping some in-depth details); Appendix B.2, B.3

The Structure of a Compiler

- a processor understands only its own machine language
- a compiler is typically a large piece of software to translate from a HLL to machine code



Front and Back Ends

The front end is dependent on

- the high level language
- the grammar

The front end should not need to know any properties of the target architecture (though exceptional cases often occur)

The back end is dependent on the target architecture.

- It knows the IR (intermediate representation)
- It knows the target's machine code
- It does not know the original HLL and grammar

Goals

- **Correctness**
  - ➔ machine code must implement the semantics of the statements EXACTLY

- **Efficiency**
  - time
  - space
  - the translation is a one-to-many mapping

Use as example:

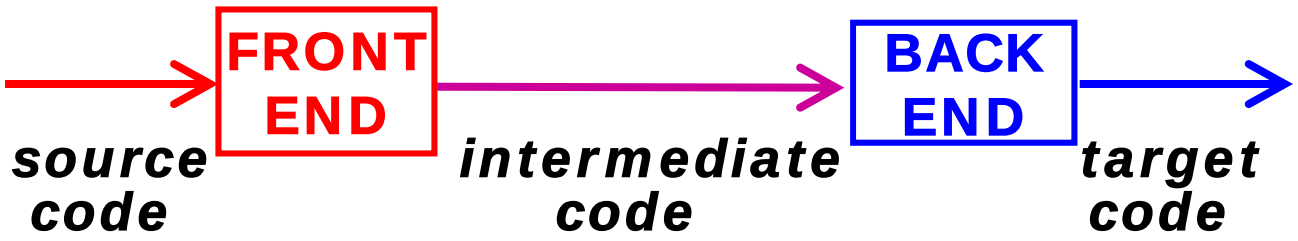
```
while(save[i] == k)
      i += 1;
```

The Structure of a Compiler

Organized as several translation steps, typically:

1. **Lexical Analysis (scanning)**
2. **Syntactic Analysis (parsing)**
3. **Semantic Analysis (checking & attaching meaning)**
4. **Optimization**
5. **Code Generation**

For the first 3 phases, there is a good analogy to how humans comprehend a language



PART 1: Lexical Analysis - Scanning

➔ **reads in individual characters and creates a string of tokens**
Examples of *tokens* are reserved words, names, operators, and punctuation symbols.

```
while(save[i] == k)
    i += 1;
```

In the example, the token sequence is:
"while", "(", "save", "[", "i", "]", "==", "k", ")", "i", "+=", "1".

"while" is recognized as a keyword in C,
"save", "i", and "j" are recognized as names (identifiers)
"1" is recognized as an integer constant (a number).

- ❑ **Simple character scanning to decompose the input (the high-level language) into basic elements**
- ❑ **output is a stream of tokens..... ➔ which is input to the parser**

Lexical Analysis - Features

- Scanning by a Lexical Analyzer must be efficient because of the volume of input

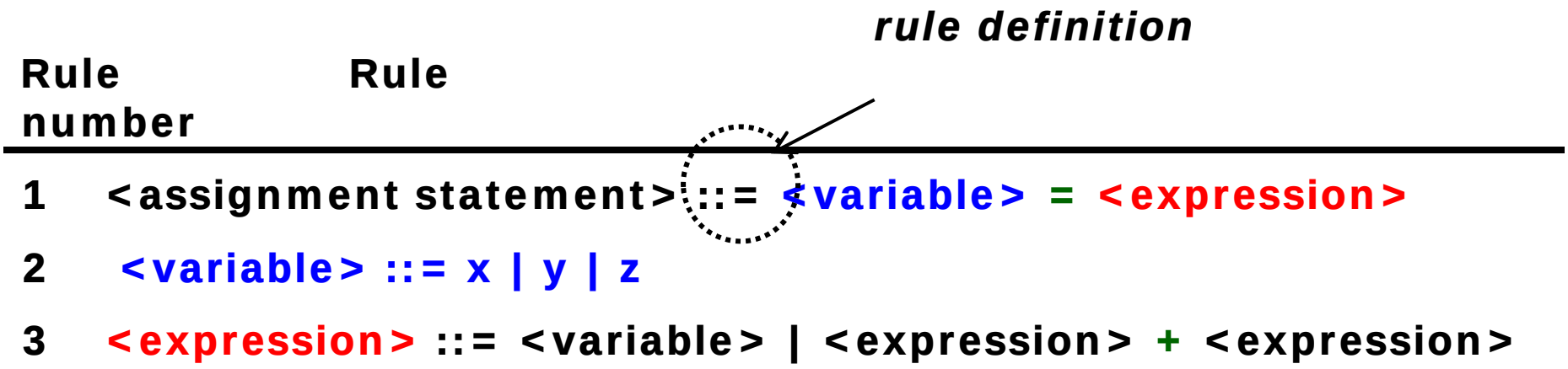
- We can list all possible tokens as an infinite set of strings
  - this set of strings defines a language
  - each string is a possible sentence in the language

- The lexical analyzer recognizes this language
- By design, the lexical language belongs to a class of languages known as the 'regular languages' because:
  - there is a well-understood and useful theory
  - they are easy to understand
  - their recognizers have efficient implementations

- Regular languages can be recognized by Finite State Automata (FSAs), also known as Finite State Machines (FSMs) or as Deterministic Finite State Automata (DFAs) NOTE: [Regular languages and FSAs are covered in CSC 320 and CSC 355.]

PART 2: Syntactic Analysis - Parsing

- The goal is to group sequences of tokens into syntactical structures as permitted by the programming language
- Given a grammar definition ➔ construct a parse tree



PART 2: Syntactic Analysis - Parsing

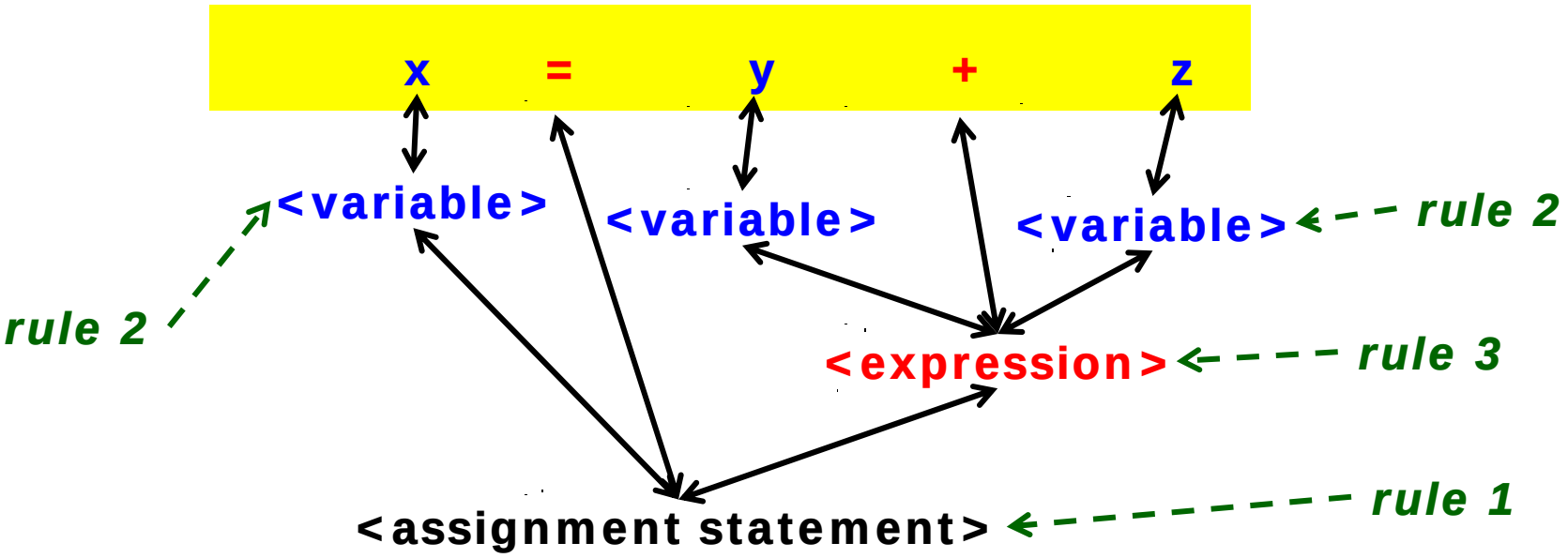
Rule number Rule

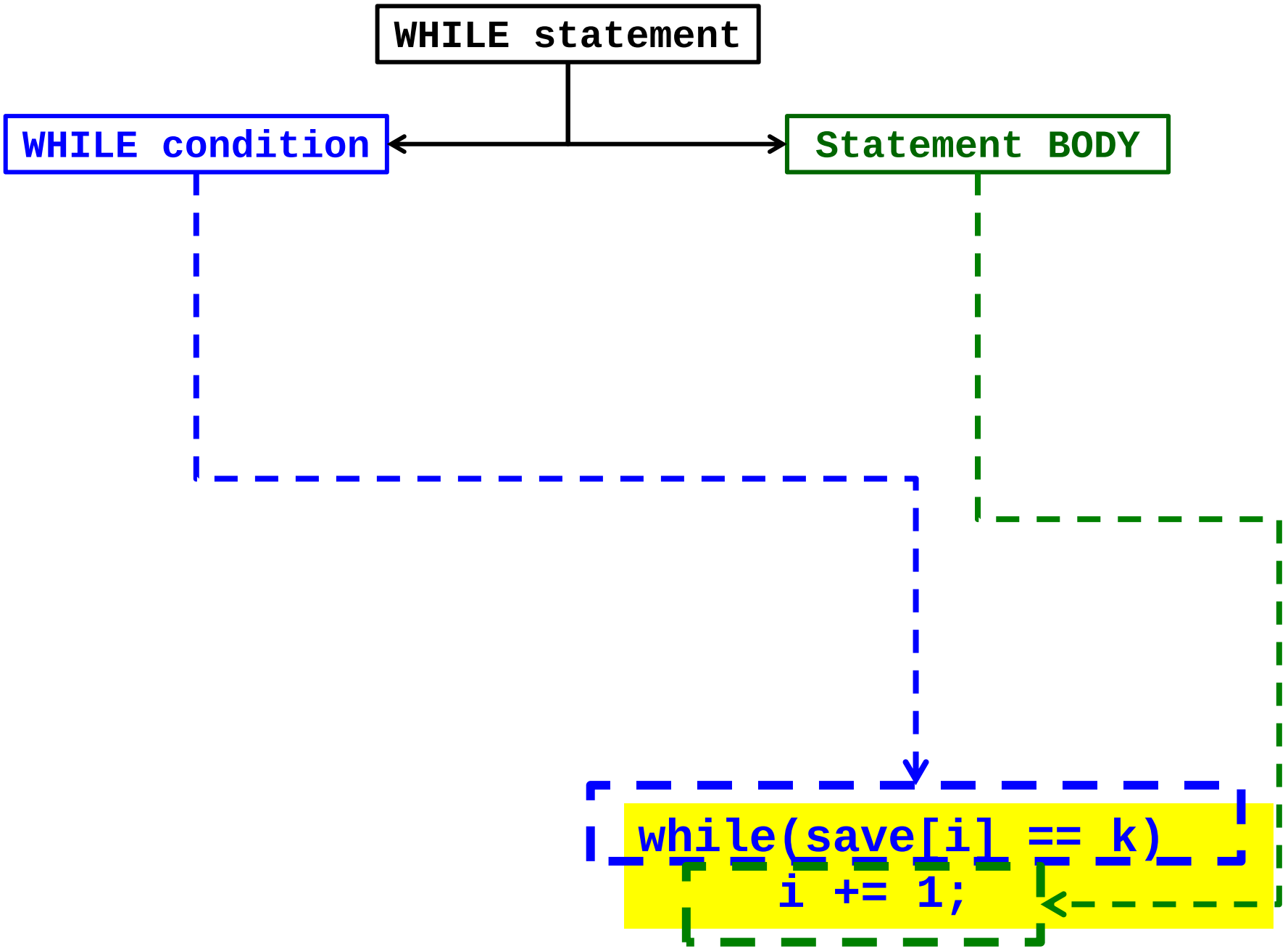
rule definition

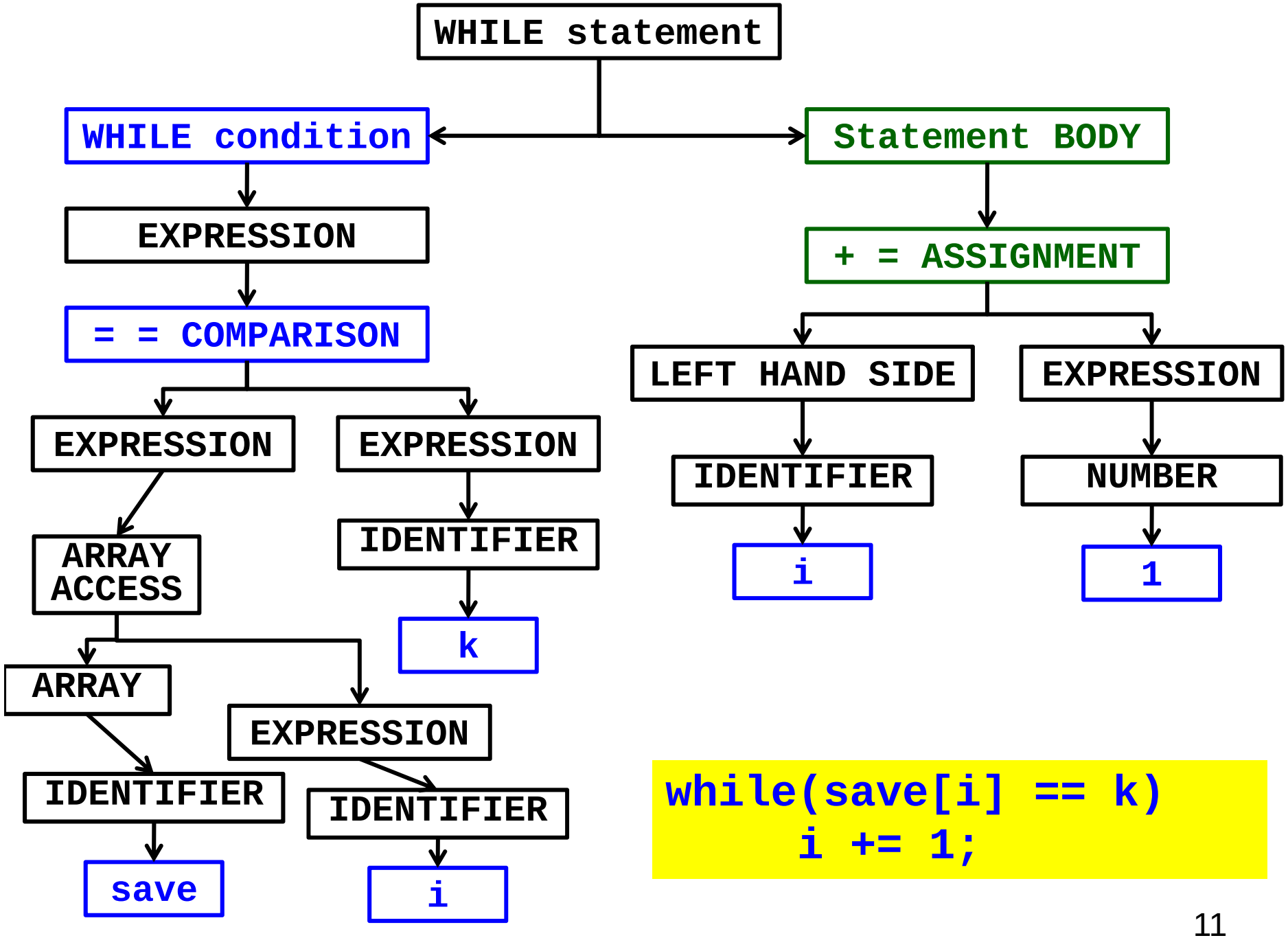
1 <assignment statement> ::= <variable> = <expression>

2 <variable> ::= x | y | z

3 <expression> ::= <variable> | <expression> + <expression>




WHILE statement
WHILE condition
Statement BODY
while(save[i] == k)
i += 1;




```
while(save[i] == k)
    i += 1;
```

PART 3: Semantic analysis – attaching meaning

- **Semantic analysis takes the abstract syntax tree and checks the program for semantic correctness**

- **Semantic analysis includes checking to ensure that:**
  - **variables and types are properly declared**
  - **that the types of operators and objects match**
    - **➔ a process called *type checking***

- **During this process, a *symbol table* is constructed, it holds information about all the named objects — e.g. classes, variables, and functions**
  - **it is used to type check the program.**

Important: this is where all the labels and variable names are listed and kept track of!

PART 4: Generation of the intermediate representation

- **As the final stage of semantic analysis, an intermediate representation form is generated from the symbol table and the abstract syntax tree.**

- **Intermediate representations usually provide simple operations on a small set of primitive types, such as integers, characters, and floats (not too different from a computer).**

- **Java bytecode represents a possible intermediate form (no registers, temporary results are kept on a stack).**

- **In modern compilers, the intermediate form may look much like a RISC instruction set but with an unbounded number of virtual registers and no limits on how far branches can jump, how big immediate constants can be, etc.**

Example of Intermediate Representation

```
# comments are written like this -- source code often
# included

# while(save[i] == k)
loop: LA R1,save          # loads the address of
                          # array 'save' into R1
      LDI R2,i
      MULT R3,R2,4        # Multiply R2 by 4
      ADD R4,R3,R1
      LDI R5,0(R4)        # load save[i]
      LDI R6,k
      BNE R5,R6,endwhileloop
# i += 1
      LDI R6, i
      ADD R7,R6,1         # increment
      STI R7,i
      B loop              # next iteration
endwhileloop:
```

The ***while*** loop example is shown using a possible intermediate representation.

The Compiler Implementer's Skill Set

- **Programming Languages Concepts**
  - Semantics of programming languages

CSC 330

- **Computer Architecture**
  - RISC & CISC machines, pipelining, cache (all for code generation)

CSC 230
CSC 350

- **Formal Language Theory**
  - Finite state machines & regular languages (lexical analysis)
  - Push-down automata & context-free languages (syntactic analysis)
  - Syntax-directed translation (building an abstract syntax tree)

CSC 320
CSC 355

The Compiler Implementer's Skill Set

- **Data Structures & Algorithms**
  - Linked-lists (for miscellaneous data structures)
  - Trees (for the abstract syntax tree)
  - Hash tables (for the symbol table)
  - Perfect hash functions (for the table of reserved keywords)
  - Dynamic programming (code generation, instruction scheduling)

CSC 225
CSC 226
CSC 425

- **Modern Algebra**
  - Algebra of types (for type checking)
  - Algebra of lattices (for theory of code optimization)

MATH XXX

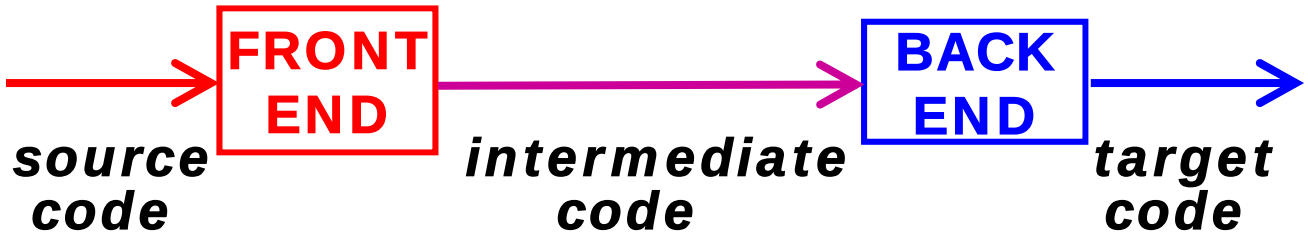
- **Pattern-matching Algorithms**
  - Tree pattern matching (code generation)

CSC 435

- **Graph Theory**
  - Graph colouring (register allocation)

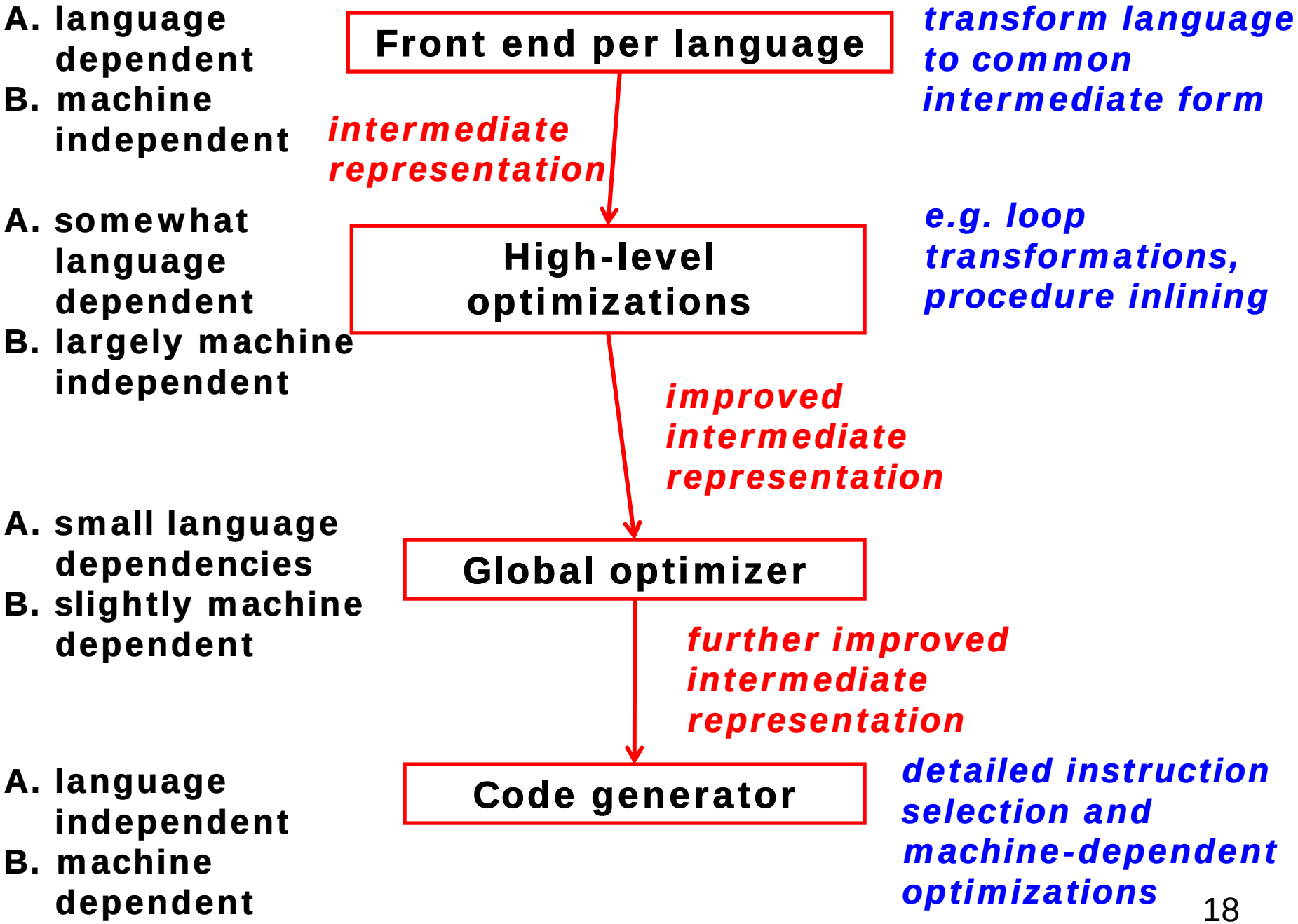
CSC 4xx

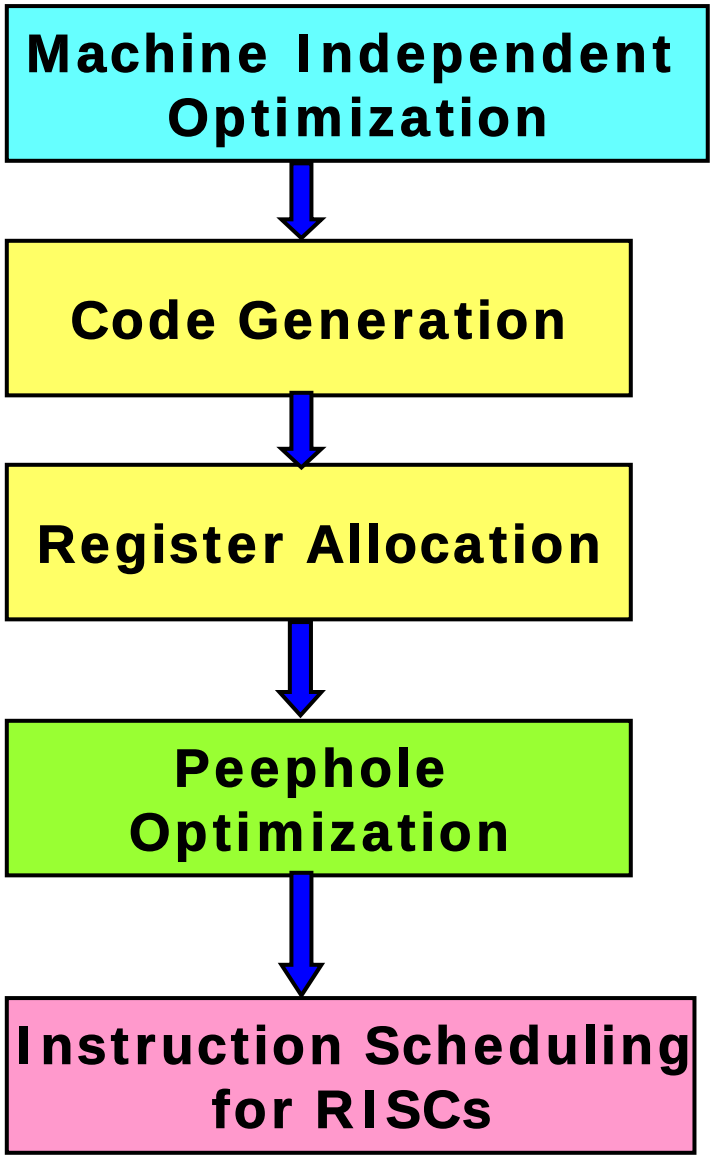
Building A Simple Compiler



Steps to Building the Front-End

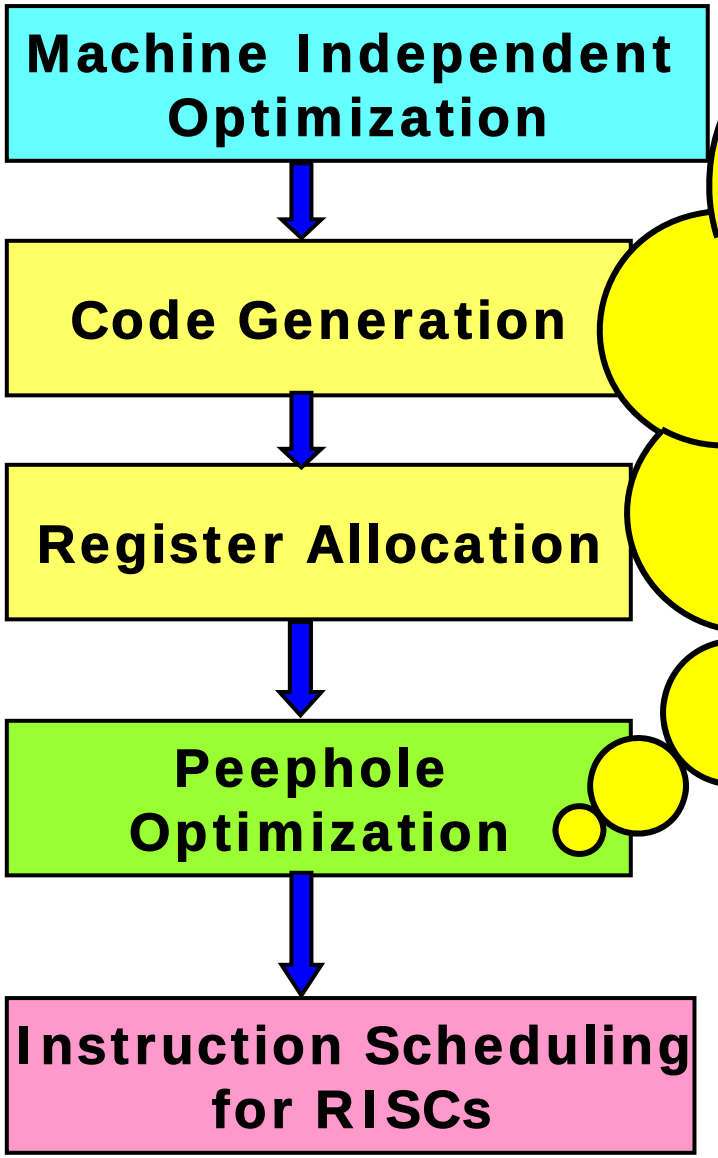
- Choose (or design) the language to be compiled.
- Understand the syntax and semantics of this language thoroughly.
- Write and debug the BNF grammar for this language.
- Implement a syntax analyzer (aka parser).
- Implement a lexical analyzer for the language.
- Integrate it with the parser and try it on test files.
- Design an abstract syntax tree structure.
- Attach semantic actions to the grammar rules so that trees are built.
- Verify that the trees are constructed properly for the test files.
- Implement tree walking functions to annotate tree nodes with datatypes and perform type checking.
- Implement tree walking functions to translate the tree into IR (intermediate representation) code.


A. language dependent
B. machine independent
Front end per language
transform language to common intermediate form
intermediate representation
A. somewhat language dependent
B. largely machine independent
High-level optimizations
e.g. loop transformations, procedure inlining
improved intermediate representation
A. small language dependencies
B. slightly machine dependent
Global optimizer
further improved intermediate representation
A. language independent
B. machine dependent
Code generator
detailed instruction selection and machine-dependent optimizations



A Typical (simple!) Compiler Back-End

An optimizing compiler can have dozens of phases and can re-apply the same optimizing transformations several times as new optimization possibilities are uncovered.



Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions and replacing these instructions by shorter or faster sequence.

An optimizing compiler can have dozens of phases and can re-apply the same optimizing transformations several times as new optimization possibilities are uncovered.

Some Examples of Machine-Independent Optimizations

Constant Propagation and Constant Expression Folding

```
n = 27;                    n = 27;
…                  ⟶       …
m = n + 1;                 m = 28;
```

Common Sub-Expression Elimination

```
x = a*b + c;               T0 = a*b;
…                  ⟶       x = T0 + c;
y = a*b - c;               …
                           y = T0 - c;
```

Code Motion

```
while(...) {               T3 = a*b/c;
 …                 ⟶      while(...) {
 n = a*b/c;                 …
 …                          n = T3;
}                           ...
                           }
```

Some Examples of Peephole Optimizations

Arithmetic Simplifications

```
ldr r3,=1024          ldr r3,=1024
mul r1,r2,r3   ⟶      mov r1,r2,lsl #10
```

Combining Instructions

```
ldr r1,[r2]           ldr r1,[r2],#4
add r2,r2,#4   ⟶
```

Branch Chaining

```
    bne L5                bne L99
    …              ⟶     …
L5: bal L99           L5: bal L99
```

Register Allocation

- A very important optimization
  - Fewer STR or LDR instructions → better performance
  - RISC makes task much simpler
  - Allocation done both locally and globally, late in compilation stage
- Most register allocation algorithms are based on 'graph colouring' – trying to colour a graph using a small number of colours.

Register Allocation by Graph Colouring

- Each virtual register is a node in an “interference graph”.
  - Two nodes are connected by an edge if the two virtual registers are ever live at the same point in the code.
- Now use a “graph colouring” algorithm: find a colour for each node in the graph such that no two adjacent nodes have the same colour. (Each colour corresponds to a real register.)
- Handling situations where there are too few registers to colour the nodes requires insertion of “spill code”.
- Complications occur when particular registers must be used (e.g. R0 for a function result) or cannot be used (e.g. destination register in an ARM multiply.)

Program Performance and Compiler Optimization

Given a "sort" routine, here is impact of compiler optimization on performance, compile time, clock cycles, instruction count, CPI (clocks per instruction)

GCC Opt. level	relative performance	clock cycles (millions)	instr. Count (millions)	CPI
none	1	158,615	114,938	1.38
1 (medium)	2.37	66,990	37,470	1.7
2 (full)	2.38	66,521	39,993	1.66
3 (proc. integr.)	2.41	65,747	44,993	1.46

Separate Compilation

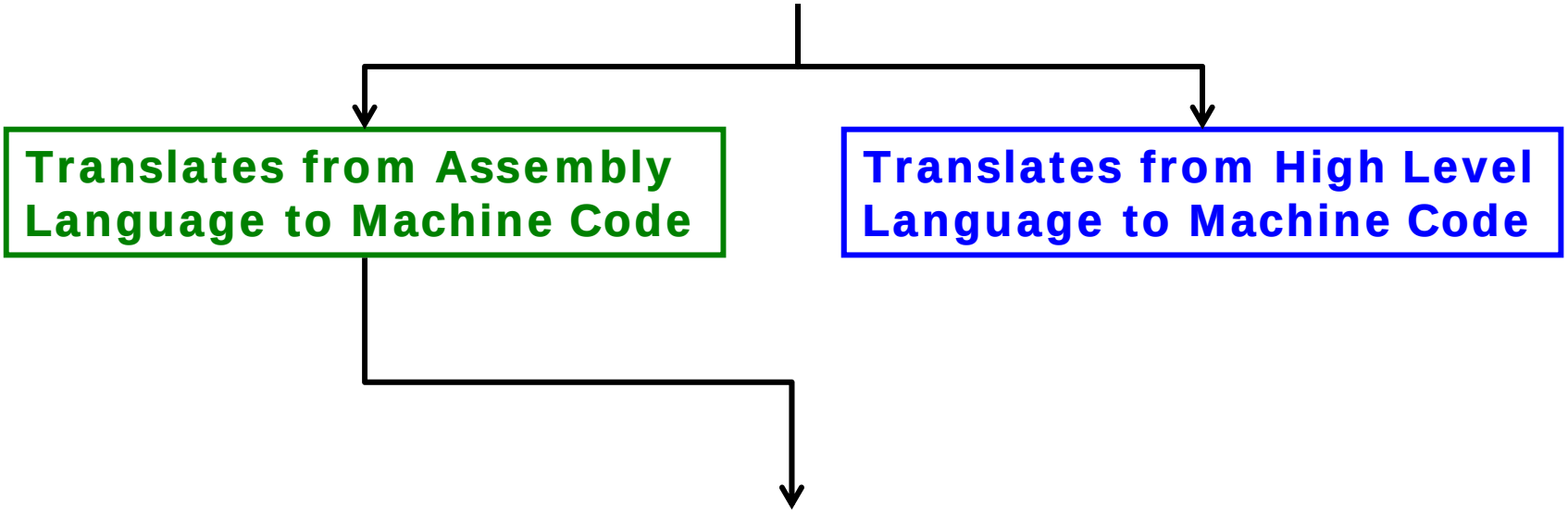
- A program is normally composed of many pieces (files).
- Some files are provided by the programmer.
- Other files provide useful subroutines (e.g. I/O, math functions, etc.) that could be used in many programs.

Compiling the whole program from source files each time something is changed is inefficient.

The solution is to translate source language files into relocatable binary files, also known as object code files (.o files on Unix).

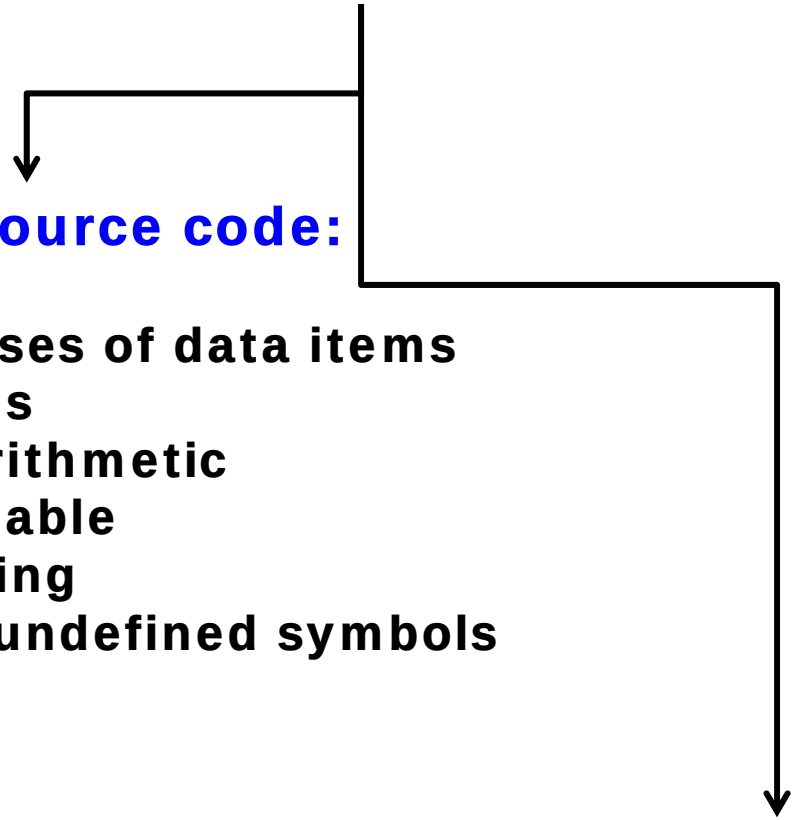
The relocatable binary files are combined by the Linker to create an executable file (the a.out default file on Unix, or a .exe file on Windows, or a .elf file on Linux).

Assembler versus Compiler



- One-To-One correspondence between one assembly language instruction and one machine instruction
- Pseudo instructions are used for convenience, e.g.,
  LDR R1, =0x123456 ⟶ LDR R1, [PC, #??]
- Macro routines can be instantiated

Two-Pass Assembler



Pass 1 over the source code:

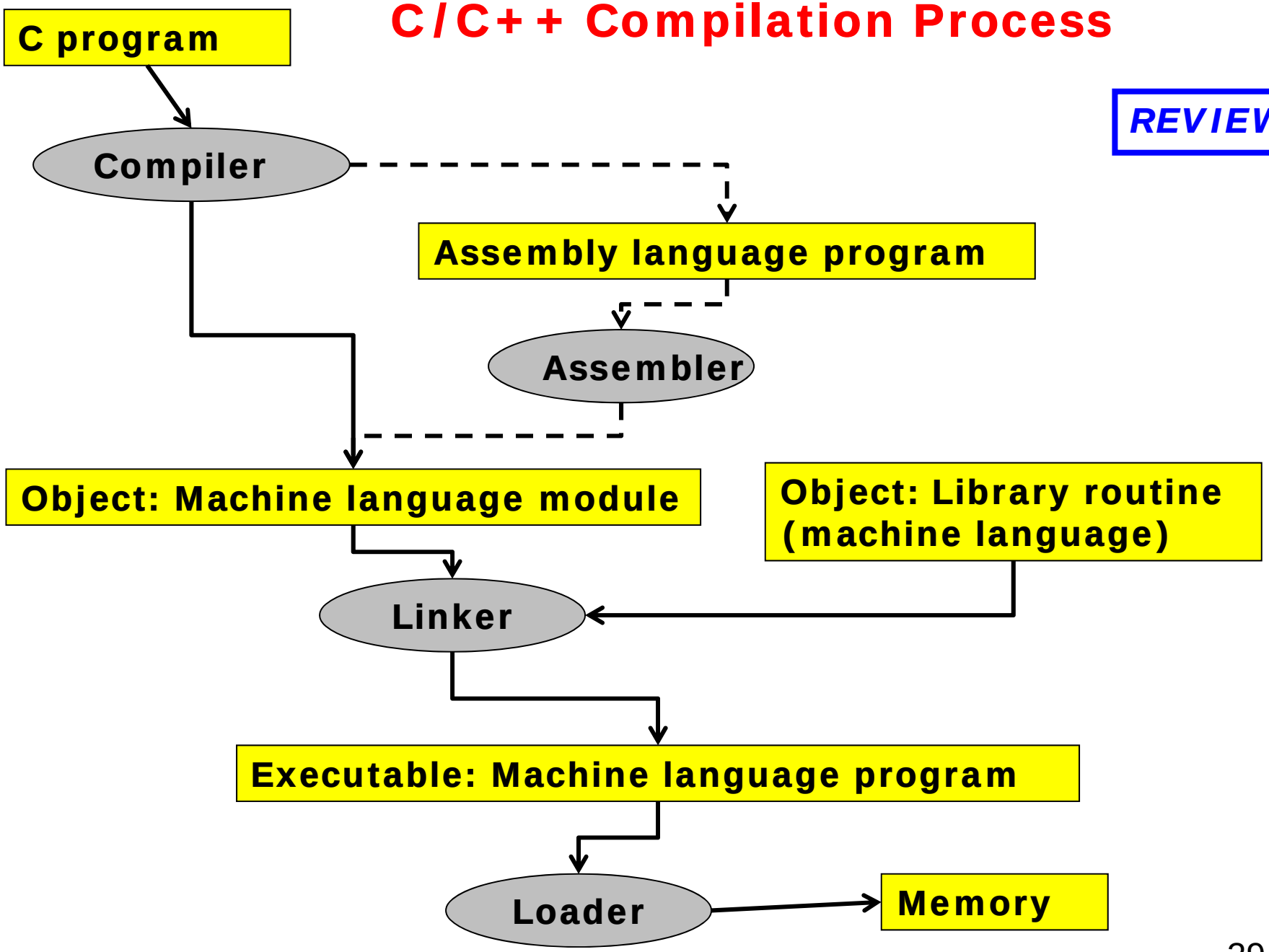
- determine addresses of data items
- select instructions
- assembly-time arithmetic
- produce symbol table
- forward referencing
- signal errors for undefined symbols

Pass 2 over the source code:

- resolve forward references
- generate machine code

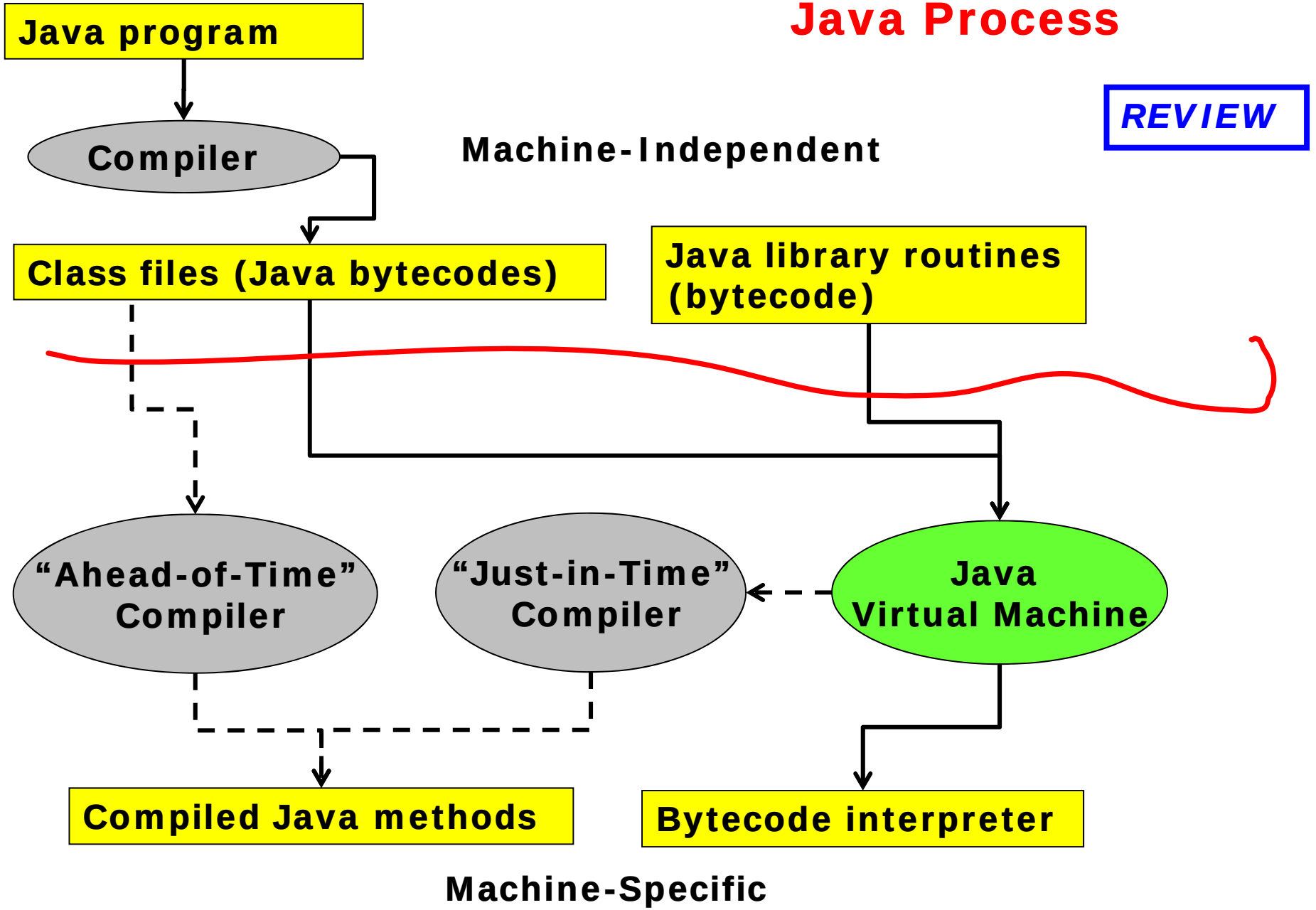
C/C++ Compilation Process

REVIEW

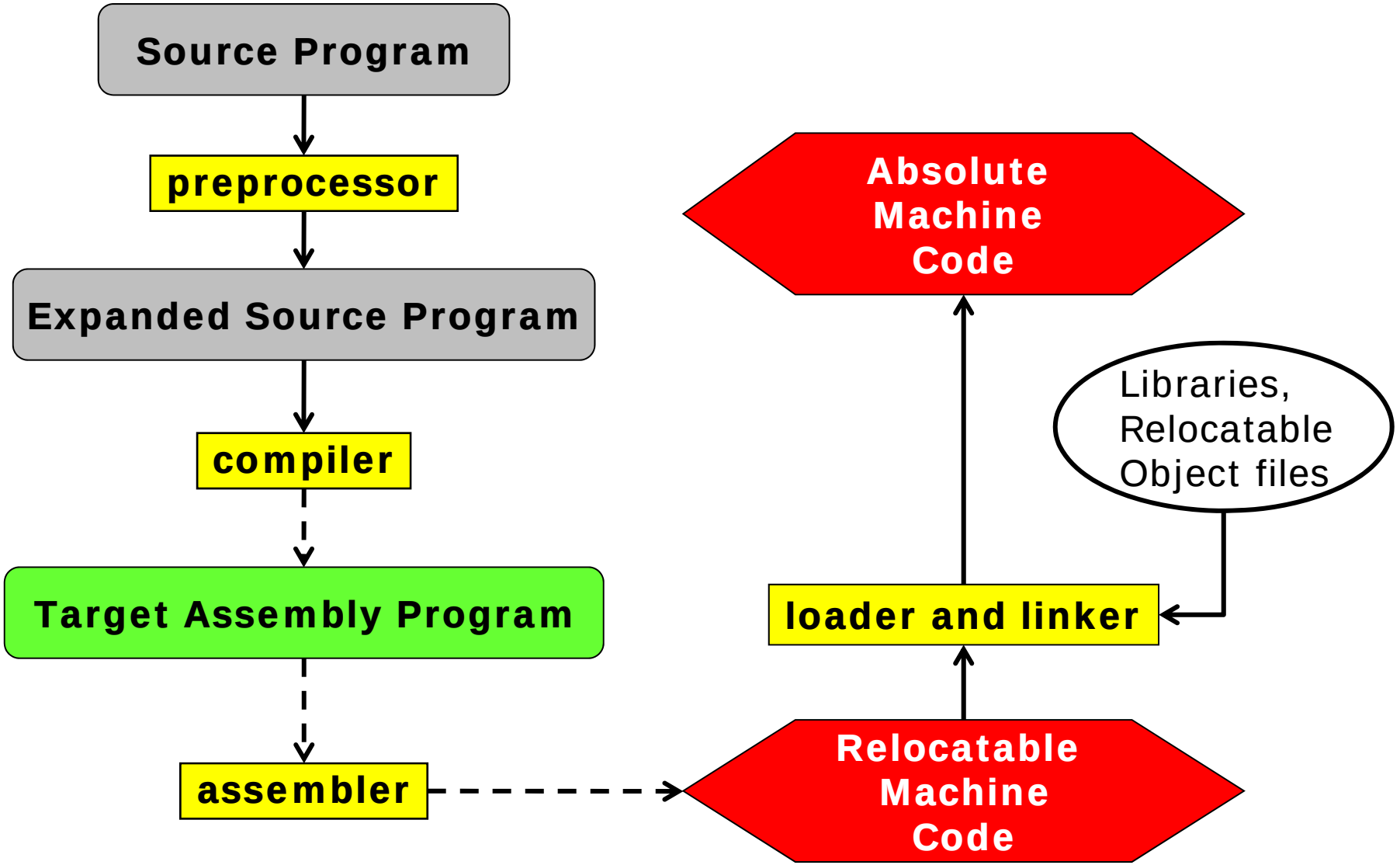


Java Process

REVIEW



Typical HLL Processing System



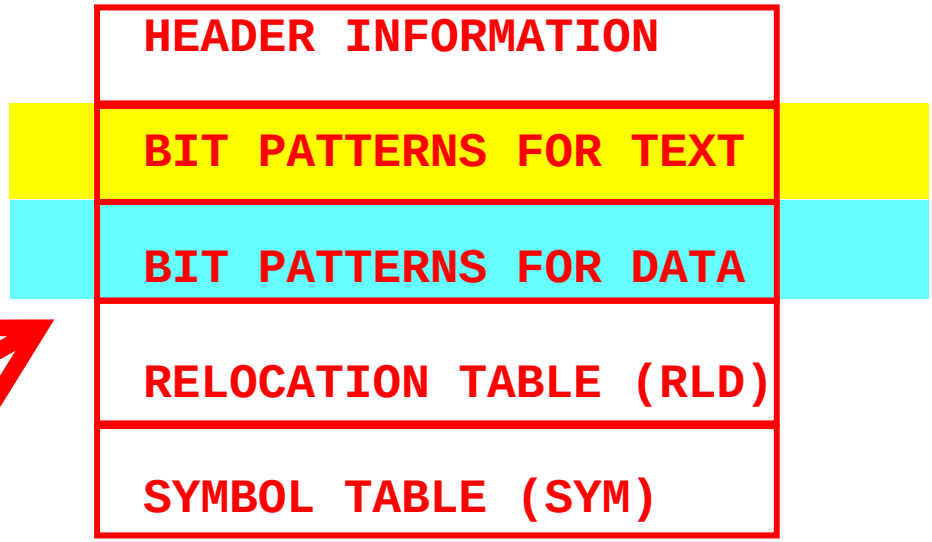
Relocatable Binary Files

.s file

```
        .text
        .global _start
        .extern _pi
        .extern _sin
_start:
        ldr r1,=_pi
        ldr r1,[r1]
        ldr r2,=X
        …
        ldr r10,=_sin
        bl  r10
        …
        .data
X:      .word 99
Y:      .word X
        …
```

assemble

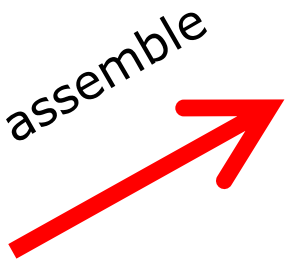
.o file



Relocatable Binary Files

.s file

```
        .text
        .global _start
        .extern _pi
        .extern _sin
_start: ldr r1,=_pi
        ldr r1,[r1]
        ldr r2,=X
        …
        ldr r10,=_sin
        bl  r10
        …
        .data
X:      .word 99
Y:      .word X
        …
```



.o file

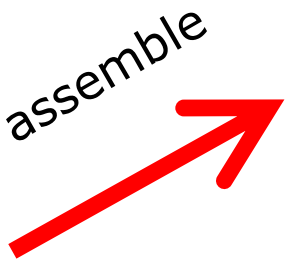
HEADER INFORMATION
BIT PATTERNS FOR TEXT
BIT PATTERNS FOR DATA
RELOCATION TABLE (RLD)
SYMBOL TABLE (SYM)

RLD: Records positions in the Text and Data parts which need to be 'fixed up' by the load point address – e.g. the contents of the word at Y will need to be adjusted when the program is loaded.

Relocatable Binary Files

.s file

```
        .text
        .global _start
        .extern _pi
        .extern _sin
_start: ldr r1,=_pi
        ldr r1,[r1]
        ldr r2,=X
        …
        ldr r10,=_sin
        bl  r10
        …
        .data
X:      .word 99
Y:      .word X
        …
```



.o file

HEADER INFORMATION
BIT PATTERNS FOR TEXT
BIT PATTERNS FOR DATA
RELOCATION TABLE (RLD)
SYMBOL TABLE (SYM)

SYM: Records exported symbols (_start) and where they occur, plus references to symbols in other files (_pi and _sin).

Linking

One or more relocatable binary files (.o files) are combined by the linker to form a single executable file. The new file has a *very similar* format to a .o file.

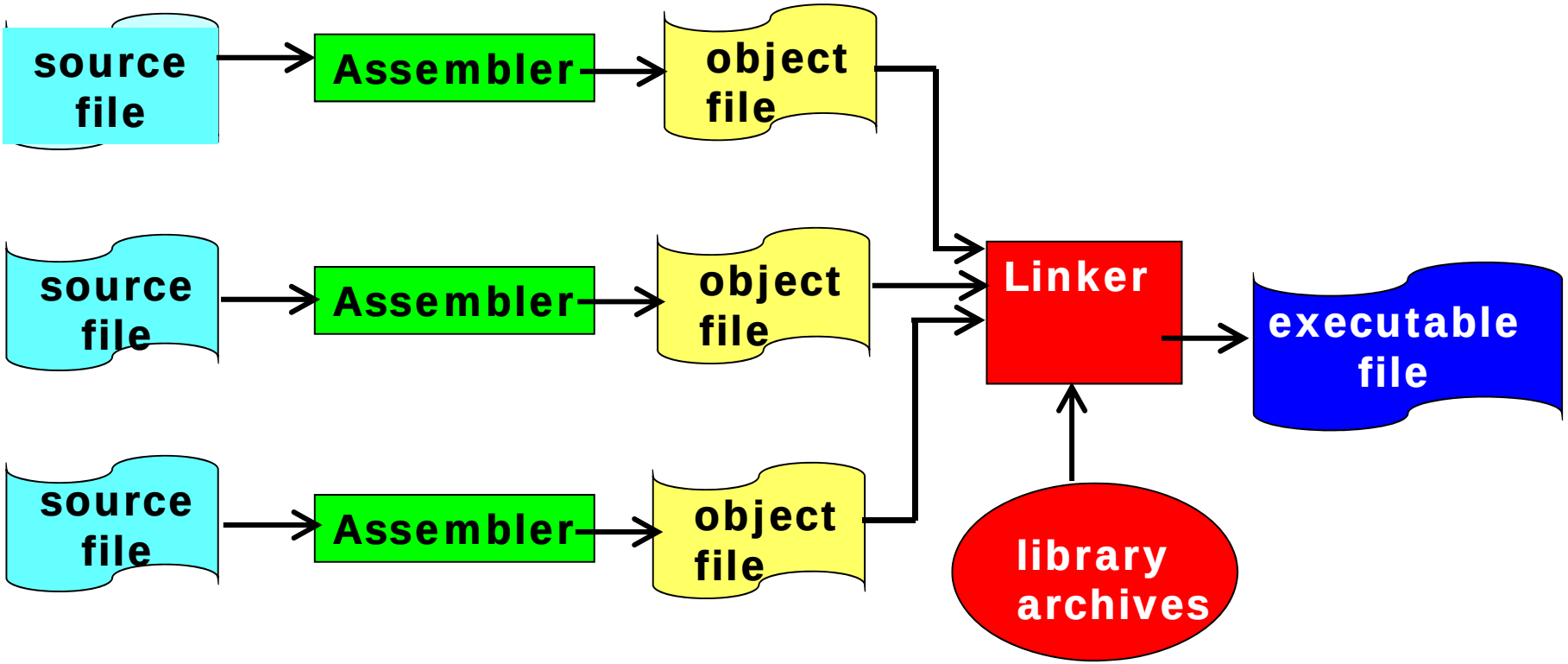
1. **The text areas of the .o files are concatenated to form a single text area.**
2. **The data areas of the .o files are similarly concatenated.**
3. **External symbols for one file are matched against exported global symbols in the other files.**
4. **The linker combines the RLD (relocation dictionary) entries to form a single RLD.**
5. **The linker combines the SYM (symbol table entries).**

There is one more important service provided by the linker, namely, the library function search ...

Linking to Library Functions

CHOICE 1: Static Linking

- When the linker has combined the .o files, it checks whether there are any unresolved external references.
- If an unresolved symbol (say _sin) exists, the linker searches libraries of precompiled functions to find one which defines that symbol. On Unix, it might find _sin defined in the sin.o member of the library file named /usr/lib/libm.a.
- That causes the linker to merge sin.o with the executable file. (It is possible that sin.o itself contains external symbols and these are added to the list of symbols that the linker needs to resolve.)
- A file like /usr/lib/libm.a is a *library archive*. It is simply a collection of .o files plus an extra table to help the linker search for symbols quickly.



Library functions are linked statically, meaning ...

- **library subroutines become part of the executable**
- **many programs contain copies of the same subroutines**

Loader

Loader: an operating system program that copies an executable file into main memory ready for execution

Typical steps performed by the loader:

1. **Read file's header to determine size of text and data segments**
2. **Create address space large enough**
3. **Copy into memory**
4. **Use the relocation dictionary to fix up all words in memory which contains addresses relative to the program's load point.**
5. **Copy parameters of "main" onto stack (if any)**
6. **Initialize registers and set stack pointer**
7. **Use the symbol table to look up the _start address, and transfer control to that address to begin execution.**

Linking to Library Functions

CHOICE 2: Dynamic Linking

- When the linker has combined the .o files, it checks whether there are any unresolved external references.
- Each reference to an unresolved symbol (say _sin) is replaced by a reference to a tiny 'stub' routine. (We may need a separate stub routine for each unresolved symbol – this depends on the O.S.)
- The result is a *much* smaller executable file (it contains no library functions).

Now, what happens when the program is loaded and run ?

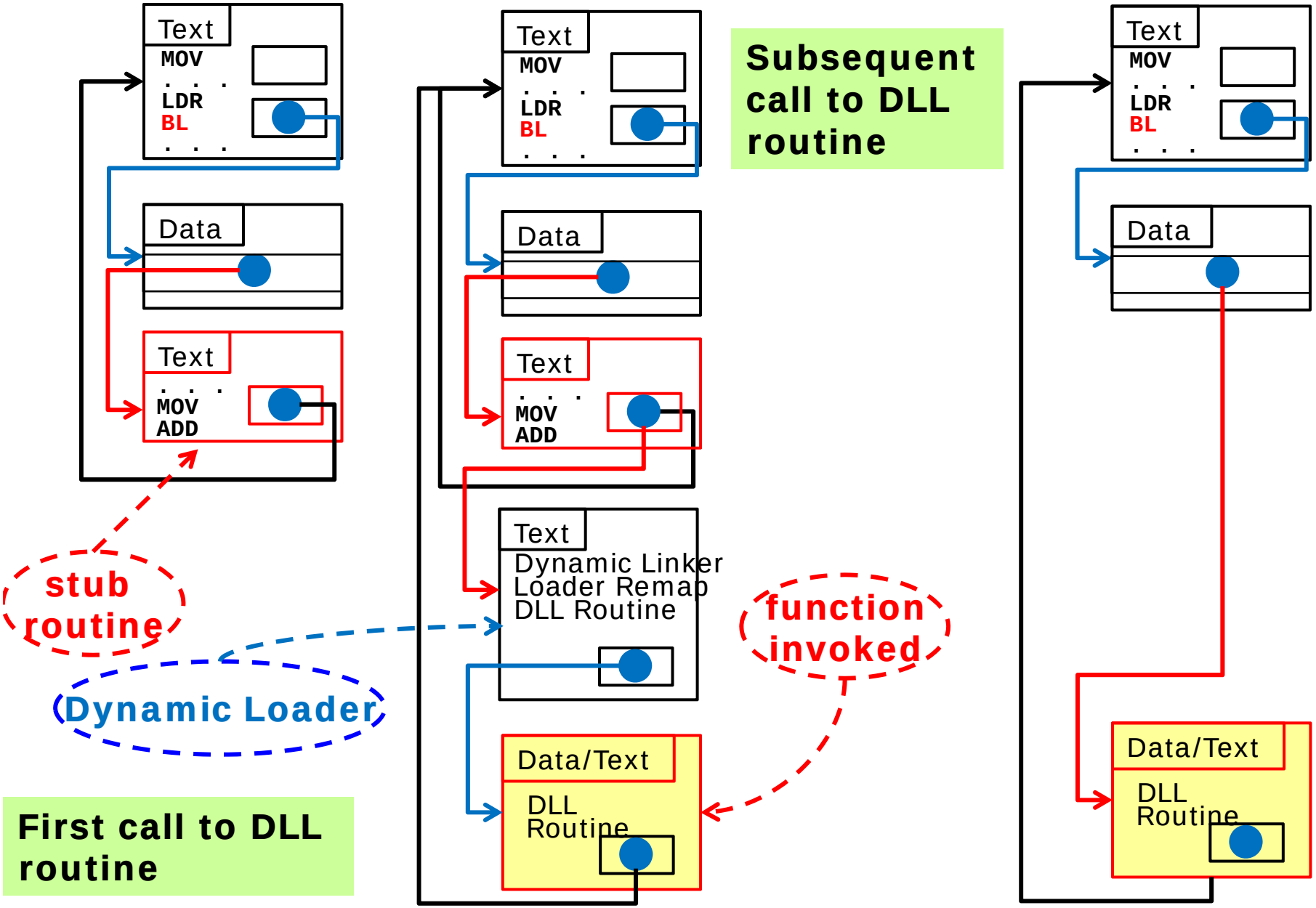
Executing a Dynamically Linked Program

The program starts up and begins execution (the same as for a statically linked program).

When the program calls a library function (e.g. _sin) for the first time:

1. **The call instruction transfers control to the stub routine.**
2. **The stub routine uses the lr register value to determine where control came from and then uses the symbol table to determine which function was called.**
3. **The stub routine invokes the dynamic loader program to copy the missing function into main memory. It searches a collection of dynamically linked libraries (DLLs) known to the system at run-time to find the function.**
4. **The stub routine patches (overwrites) the address used by the call instruction so that the next call is made directly to the function, thus by-passing the stub routine.**

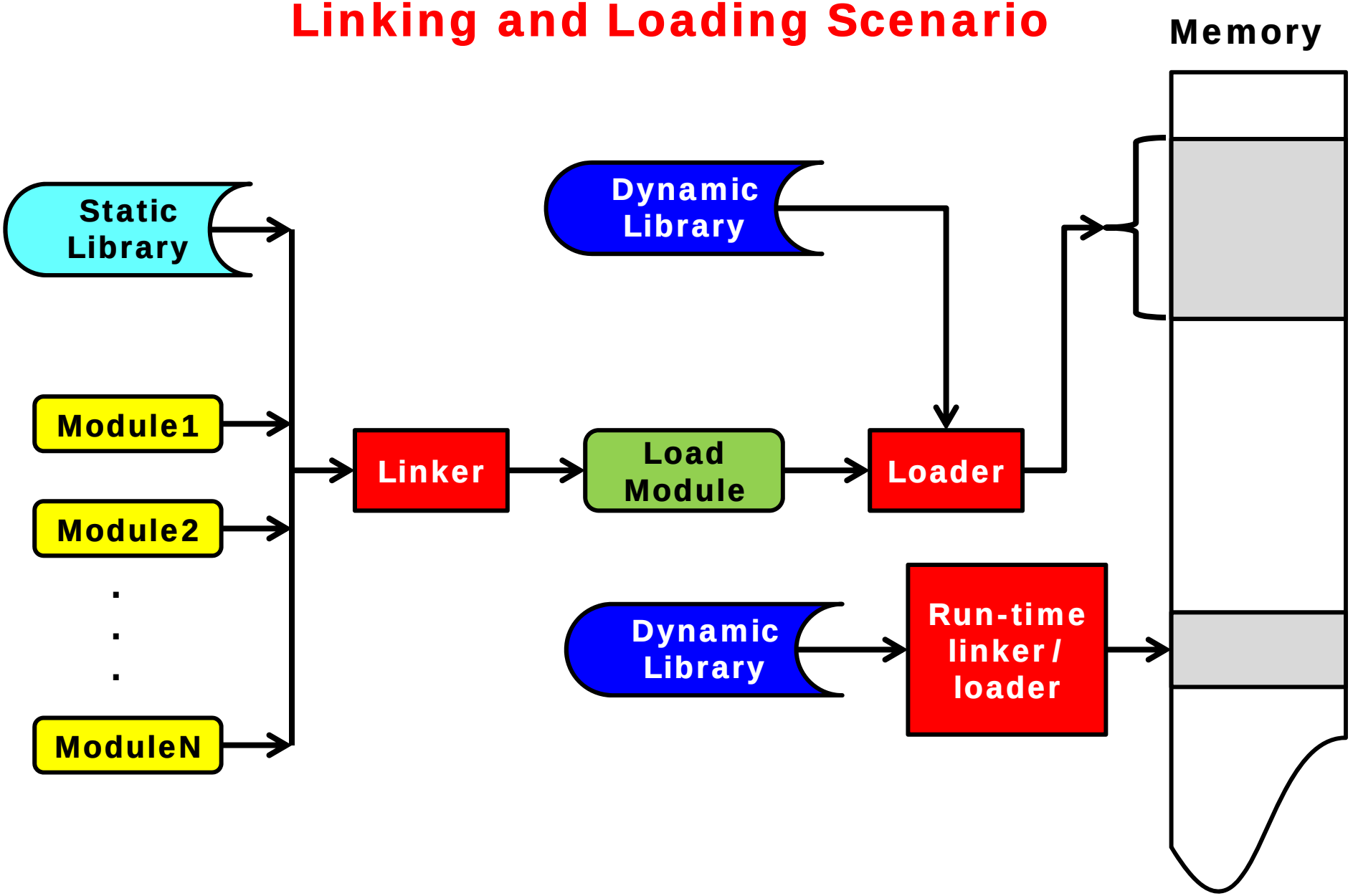
Dynamically Linked Libraries (DLL)



Final Notes about DLLs

1. **Executable files are smaller.**
2. **A program does not (usually) need to be recompiled when a bug has been fixed in a library function and a new DLL distributed.**
3. **When a function is dynamically loaded, it is copied into a separate region of memory that is shared between many programs.**
4. **Several active programs can all share the same copy of a function (a big saving for an O.S. like Windows).**
5. **Requires functions in DLLs to be carefully programmed so that they can be shared (e.g. no static variables – use only registers and storage in the stack frame as variables).**
6. **DLL hell (search for this term with Google!) occurs when there are multiple versions of a DLL available and some programs work only with an old version while others work only with a new version.**

Linking and Loading Scenario



What is a cross compiler?

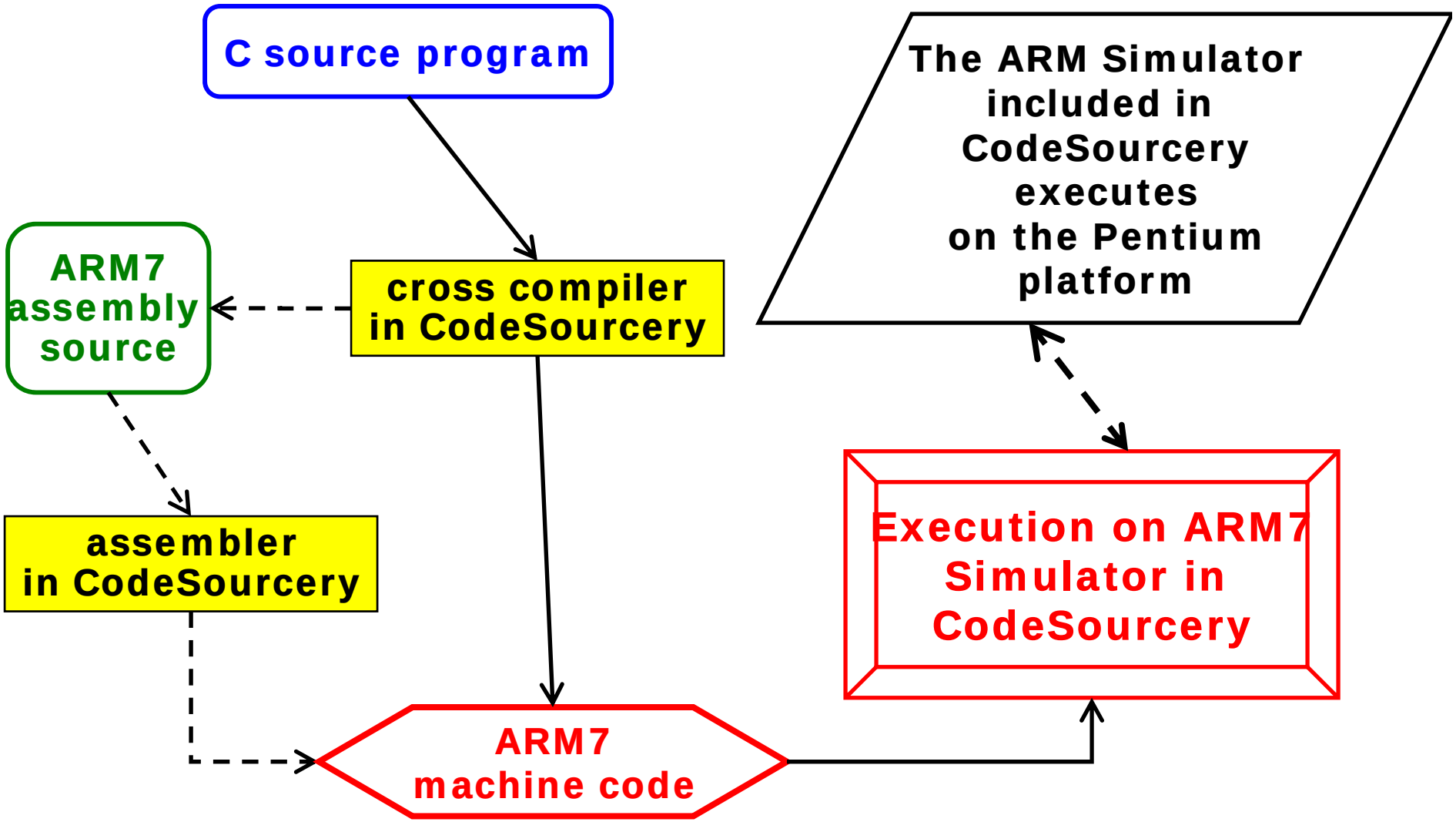
A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the cross compiler itself is executing on

It is handy to compile code for a platform for which there is no access

- **e.g. with embedded systems: here the target hardware may not have the resources or capabilities to compile code on its own**

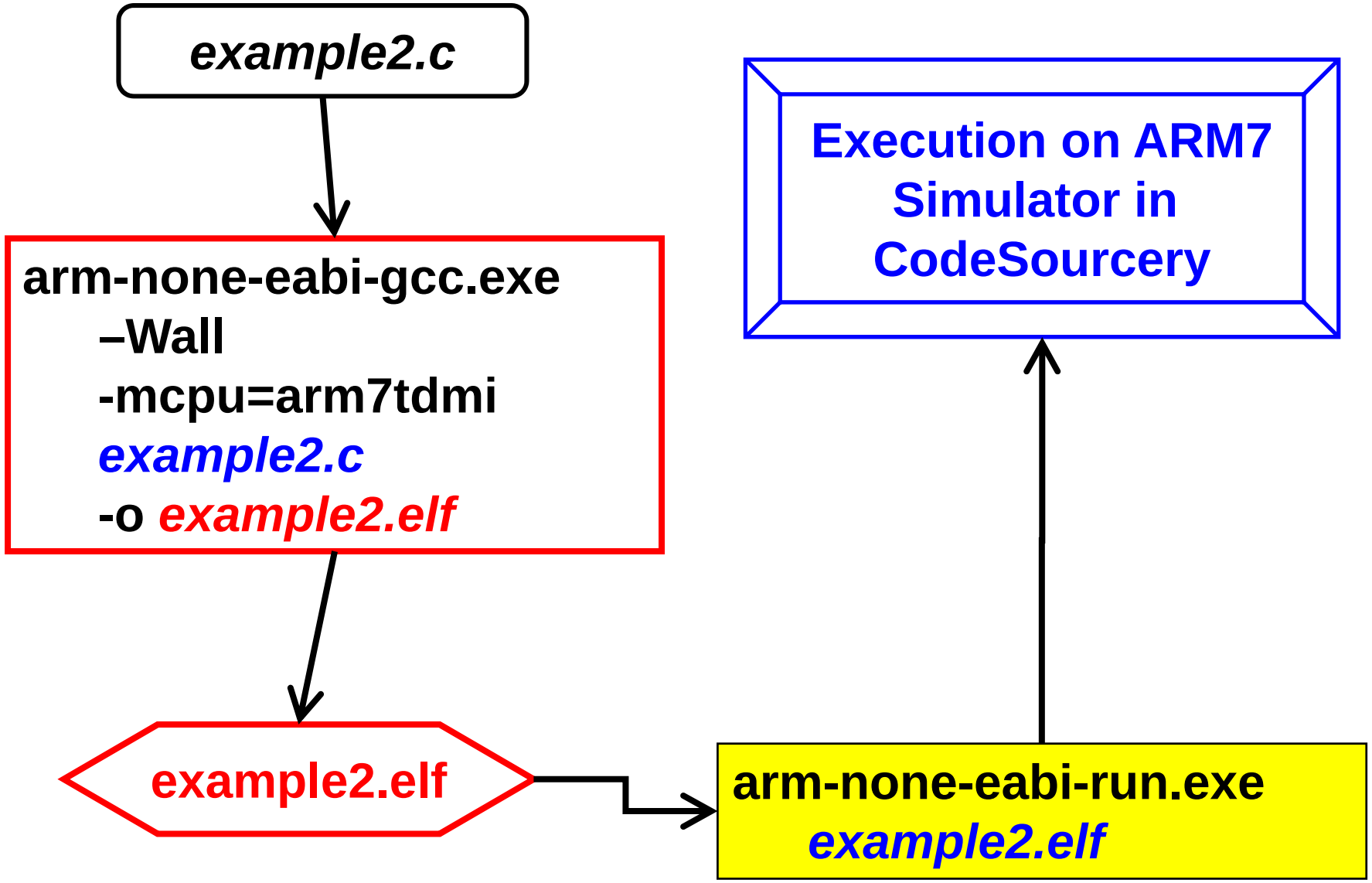
Cross-compilation is typically more involved and prone to errors than with native compilation.

C Language Processing System (future and simplified)



This will allow programming in C and execution on an ARM platform

Possible use of a cross compiler



Use of a cross compiler in assignment 4