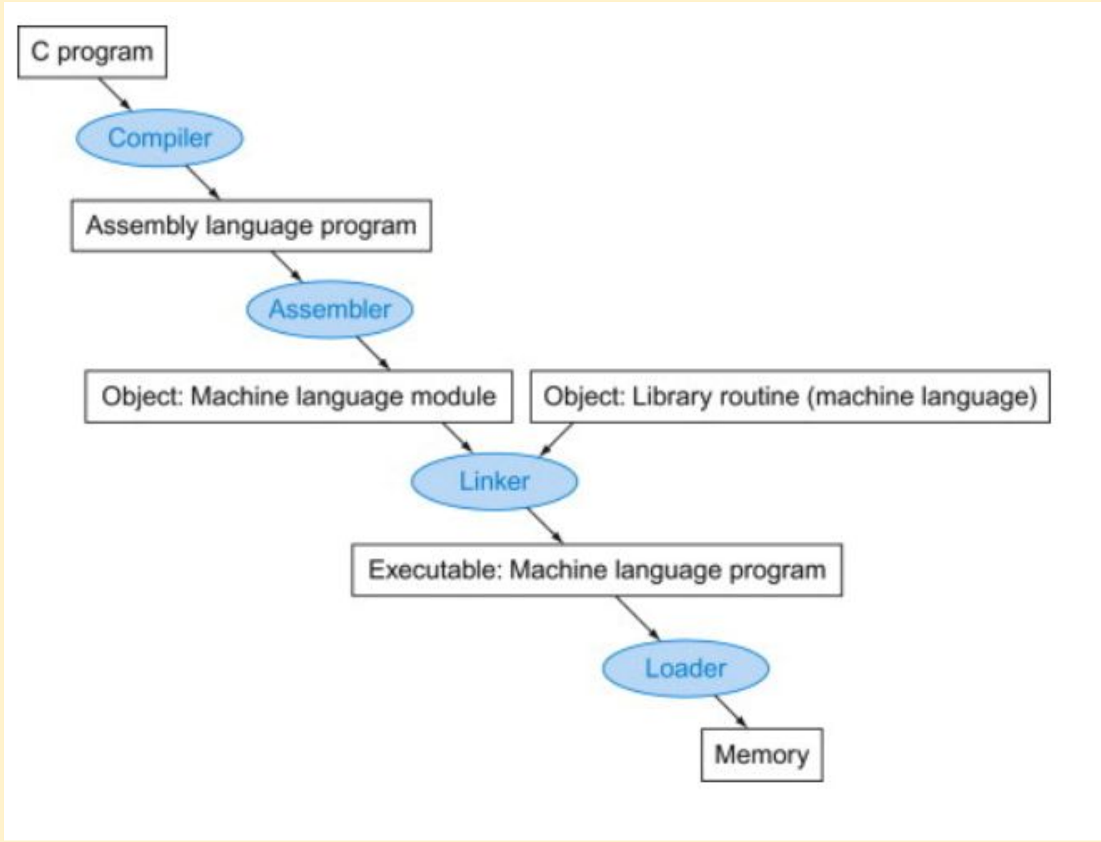
Compile C

4 steps:

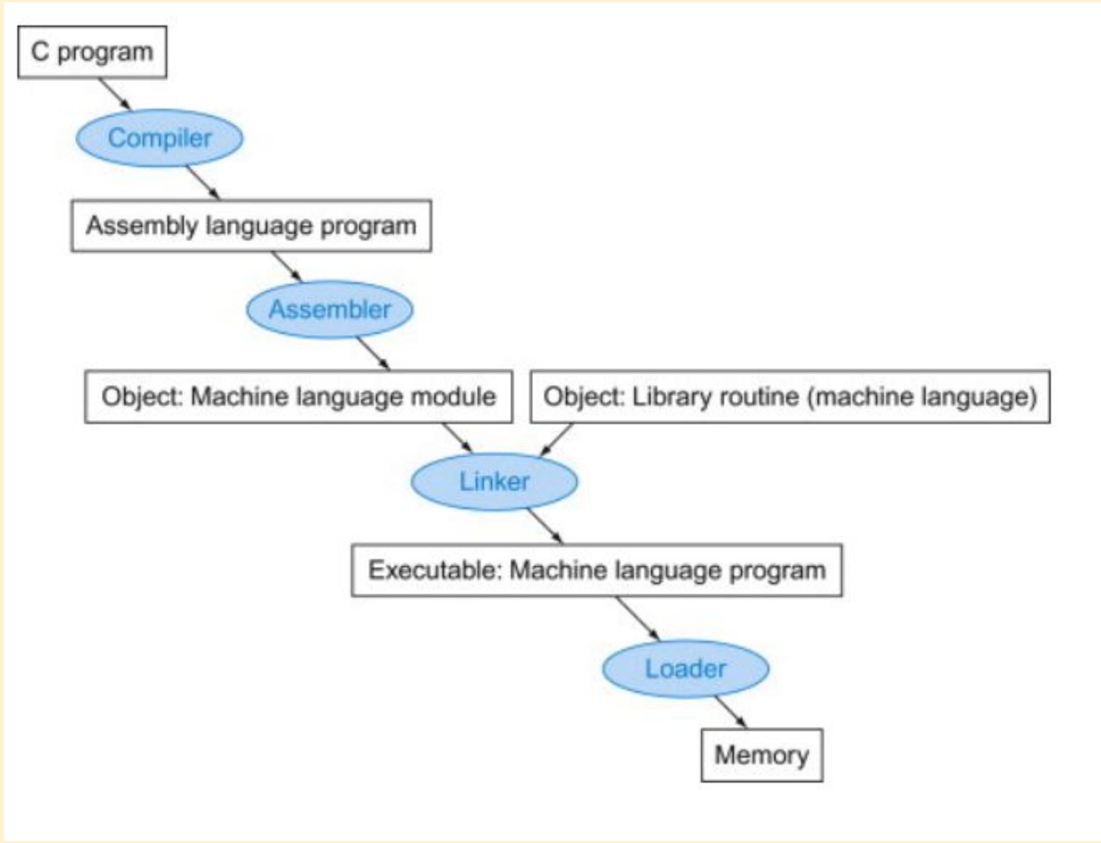
1. the <u>compiler</u> translates C into assembly language
2. the <u>assembler</u> translates assembly code (and pseudoinstructions) into machine code (object files); the assembler creates a *symbol table* to match labels to addresses.
3. all object files are linked by the <u>linker</u> into one executable
4. the <u>loader</u> loads the executable into memory


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Linking

Steps:

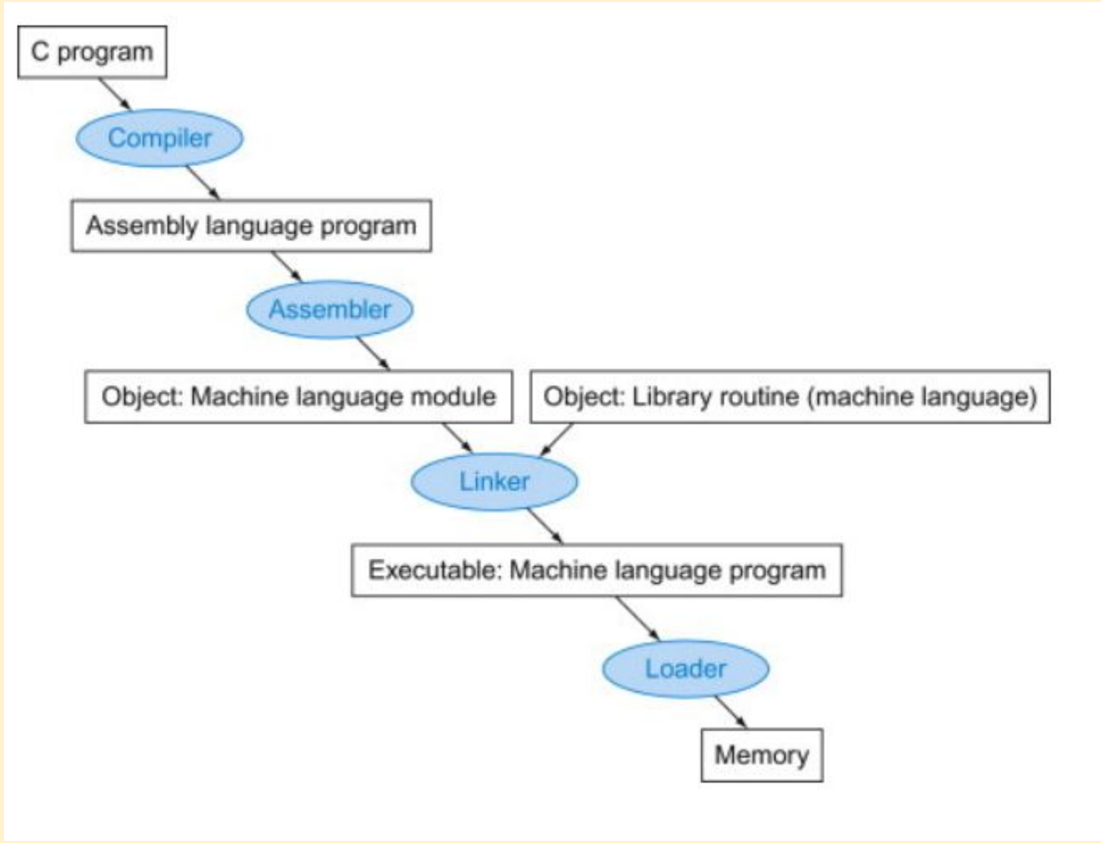
- place code and data modules symbolically in memory
- determine the addresses of data and instruction labels
- patch both the internal and external references


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Loading

Steps:

- read the executable file
- create address space for the program and data
- copy the instructions and data into memory
- copy parameters (if any) to the main program on to the stack
- initialize registers and set the stack pointer
- jump to a start-up routine that starts "main" and returns control to the system upon program completion


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

DLL - dynamically linked libraries

The linking and loading steps above were traditional methods, called the static approach. Disadvantages:

- the library routines are part of the executable, if the library is updated, the executable still has the old code
- it loads all libraries that are called anywhere, bloating the program with system routines that may not actually be called during the execution

Dynamically linked libraries are a more modern approach.

DLL

Approach:

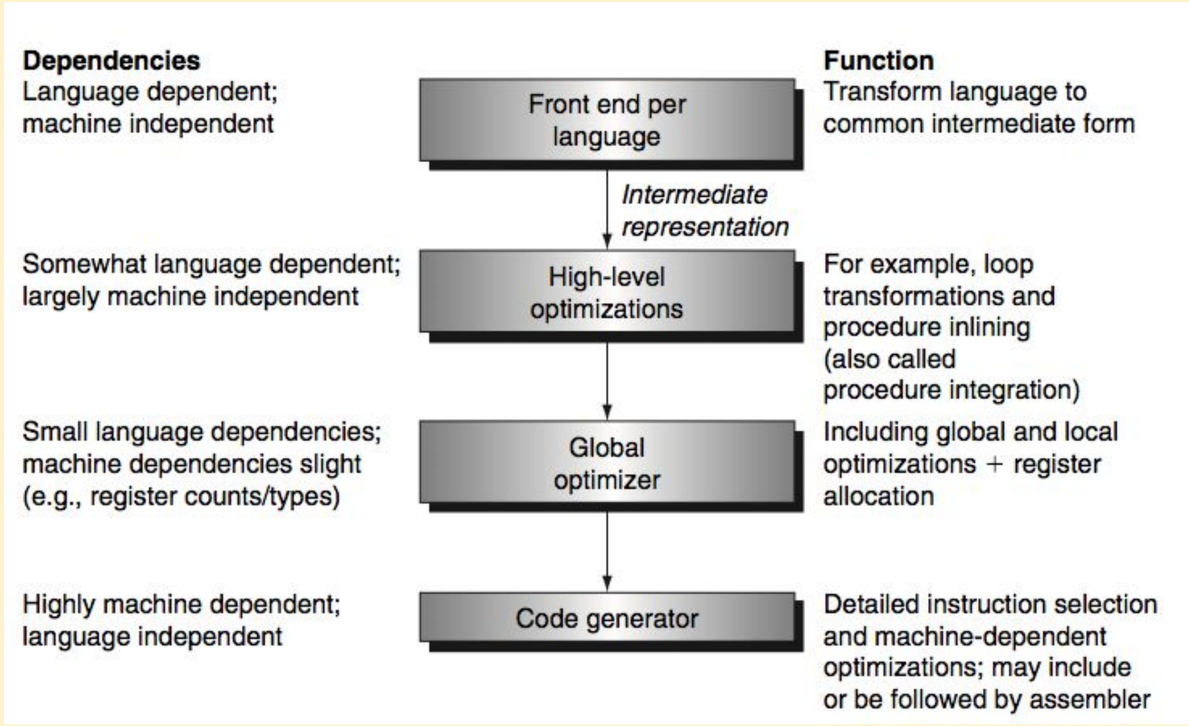
- library calls are represented by a "dummy" placeholder that points to the routine
- when a routine is called, the linker/loader finds the routine, loads it into memory
- thereafter the routine is ready to be called directly

DLL lazy loading

- Only link a routine after it is called
- Traditional DLL links (but does not load) every routine potentially called in the executable

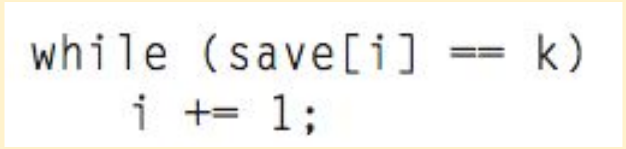
Compiling programs

- Front end translates a higher-level language to a machine-independent intermediary language
- Performs series of optimizations
- Last level generates code

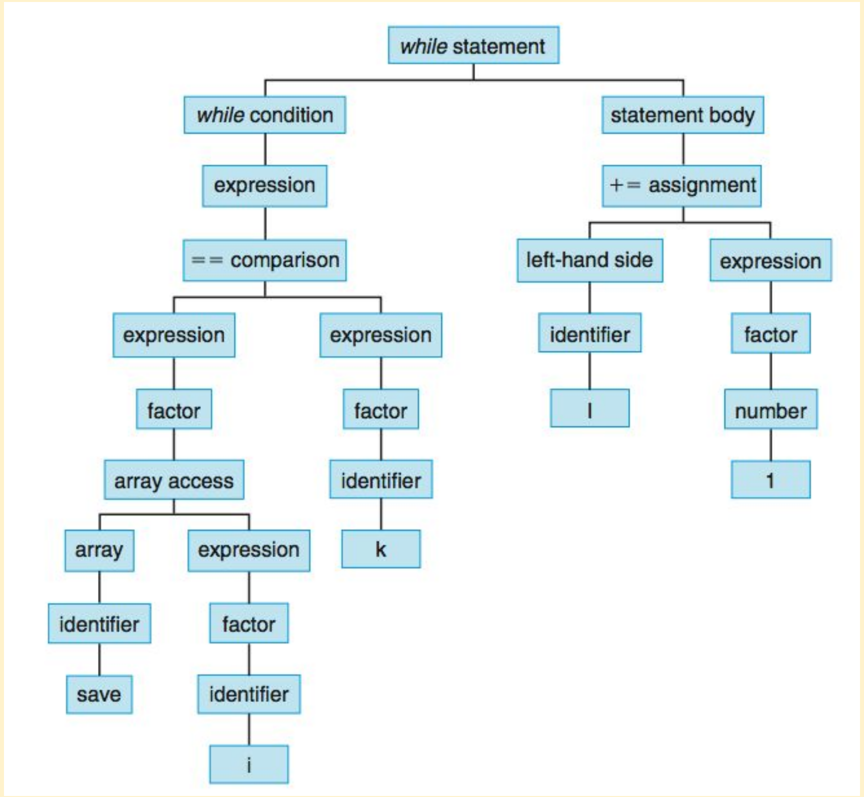

Dependencies
Language dependent; machine independent
Front end per language
Function
Transform language to common intermediate form
Intermediate representation
Somewhat language dependent; largely machine independent
High-level optimizations
For example, loop transformations and procedure inlining (also called procedure integration)
Small language dependencies; machine dependencies slight (e.g., register counts/types)
Global optimizer
Including global and local optimizations + register allocation
Highly machine dependent; language independent
Code generator
Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler

Front End

- The while statement is translated into an abstract syntax tree
- There are tokens: identifiers and operators
- Syntax rules construct the tree

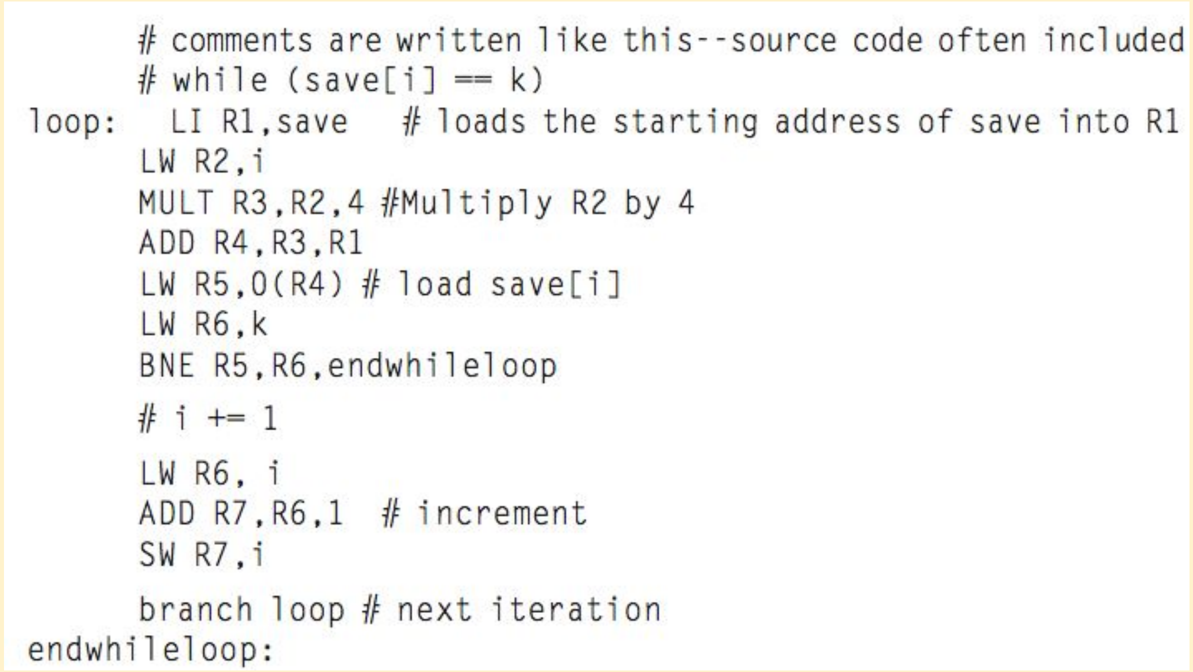


```
while (save[i] == k)
    i += 1;
```



intermediate form

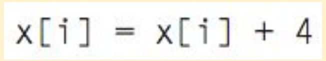
- registers are represented as unlimited Rnn registers



```
      # comments are written like this--source code often included
      # while (save[i] == k)
loop:  LI R1,save   # loads the starting address of save into R1
      LW R2,i
      MULT R3,R2,4 #Multiply R2 by 4
      ADD R4,R3,R1
      LW R5,0(R4) # load save[i]
      LW R6,k
      BNE R5,R6,endwhileloop
      # i += 1
      LW R6, i
      ADD R7,R6,1  # increment
      SW R7,i
      branch loop # next iteration
endwhileloop:
```

optimizations

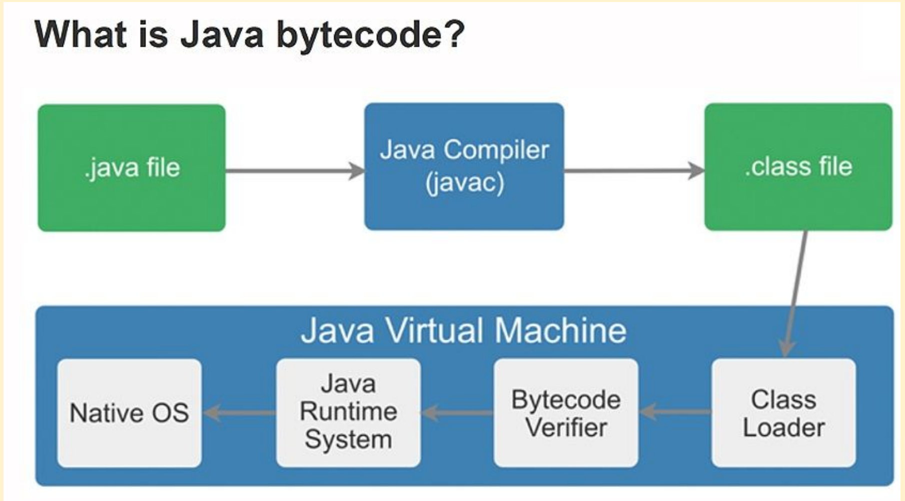
- strength reduction - replace a slow instruction like MULT with shift left
- dead store elimination - get rid of stores to values that are not used again
- dead code elimination - get rid of code that is not executed or does not affect the rest of the program
- loop unrolling: repeat loop body statements with adjusted indices to need fewer jumps
- subexpression elimination: calculate x[i] address once, not twice

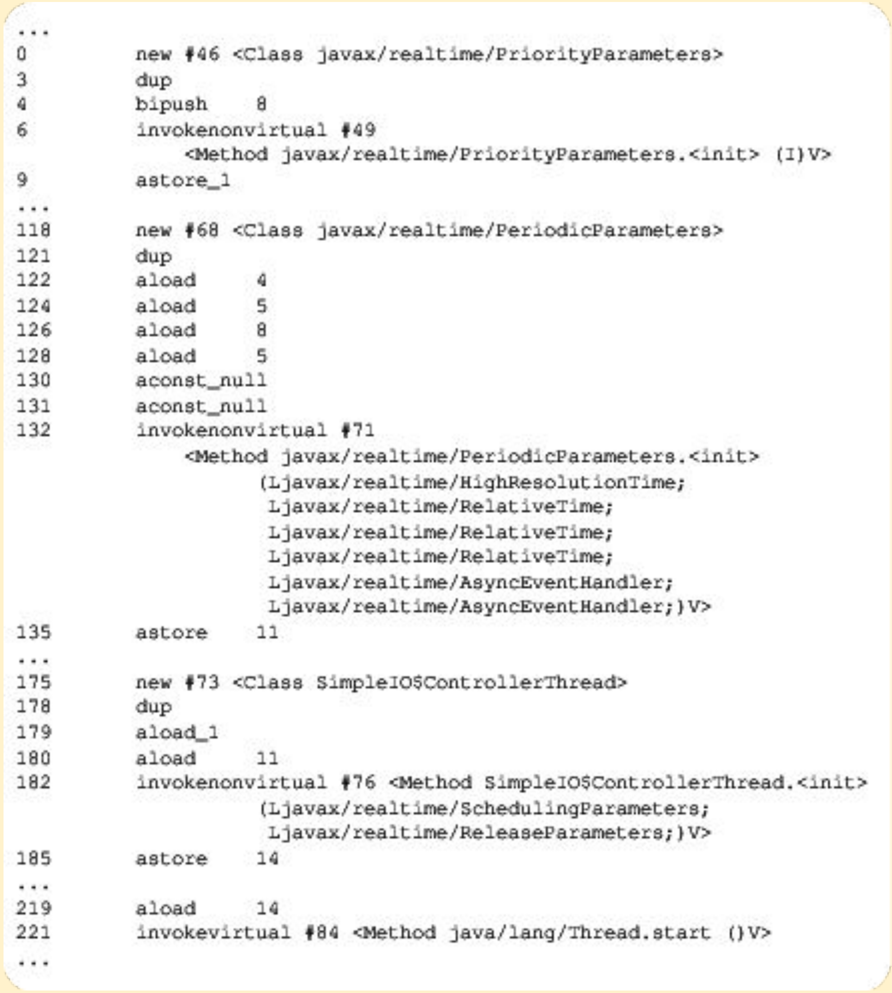


```
x[i] = x[i] + 4
```

Java

- Java is compiled into Java bytecode
- the JVM Java Virtual Machine executes the bytecode





```
...
0          new #46 <Class javax/realtime/PriorityParameters>
3          dup
4          bipush    8
6          invokenonvirtual #49
               <Method javax/realtime/PriorityParameters.<init> (I)V>
9          astore_1
...
118        new #68 <Class javax/realtime/PeriodicParameters>
121        dup
122        aload     4
124        aload     5
126        aload     8
128        aload     5
130        aconst_null
131        aconst_null
132        invokenonvirtual #71
               <Method javax/realtime/PeriodicParameters.<init>
                       (Ljavax/realtime/HighResolutionTime;
                        Ljavax/realtime/RelativeTime;
                        Ljavax/realtime/RelativeTime;
                        Ljavax/realtime/RelativeTime;
                        Ljavax/realtime/AsyncEventHandler;
                        Ljavax/realtime/AsyncEventHandler;)V>
135        astore    11
...
175        new #73 <Class SimpleIO$ControllerThread>
178        dup
179        aload_1
180        aload     11
182        invokenonvirtual #76 <Method SimpleIO$ControllerThread.<init>
                       (Ljavax/realtime/SchedulingParameters;
                        Ljavax/realtime/ReleaseParameters;)V>
185        astore    14
...
219        aload     14
221        invokevirtual #84 <Method java/lang/Thread.start ()V>
...
```

Comparing Java and MIPS

Java uses a stack instead of registers for operations:

- operands are pushed on the stack, operated on, then popped off

MIPS instructions are always 4 bytes, Java bytecode instructions vary from 1 to 5 bytes because the original designers were concerned about space

Compiled vs. interpreted languages

- Compiled languages have to be recompiled for every machine, but that compiled code is optimized and fast for that machine.
- Interpreted languages don't have to be recompiled but tend to have lower performance.
- Interpreted languages like Java can be made faster by Just in Time (JIT) compilers. These compilers find blocks of code that can be optimized and then compile them into native machine code at run time.

Programming evolution

Early assembly/languages

- go-to
- global variables (our registers are global)
- functions (reduce our cognitive load)

Problems led to higher-level languages that conceal details:

- functions with scope

Problems led to further concealment:

- objects with getter/setter helpers

John Locke (b. 1632, d. 1704)

“The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three: 1. Combining several simple ideas into one compound one, and thus all complex ideas are made. 2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations. 3. The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.”

— John Locke, An Essay Concerning Human Understanding

Creativity and coding

the use of the **imagination or original ideas**, especially in the production of **an artistic work**

music	how we feel
art	what we see, feel, believe
poetry	what we imagine, feel, see, believe
math/logic	what we assert to be true
code	express a **logical procedure** in **syntax**

The idioms by which we communicate the symbols of our poetry:

- conditionals, loops, functions

Writing good code

- Should you do things like loop unrolling in your higher-level language code?
- No
- Let the compiler optimize the code

"... programs must be written for people to read, and only incidentally for machines to execute."

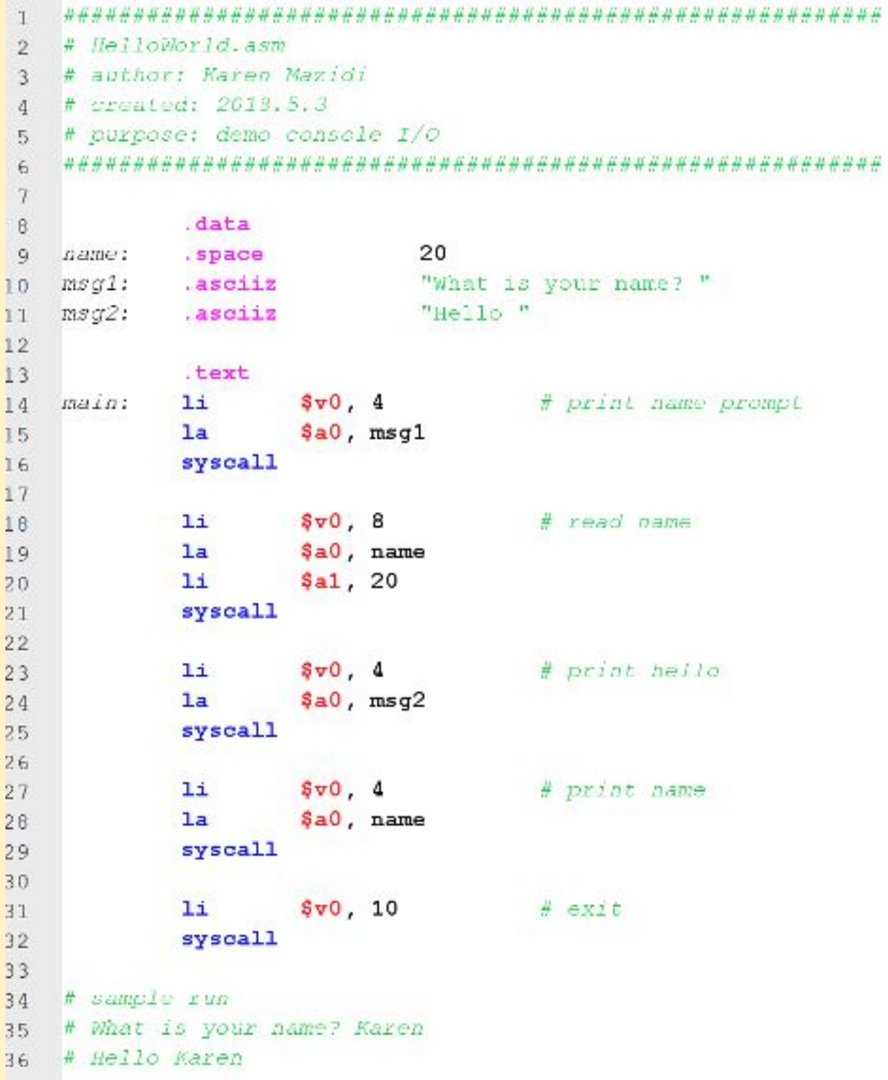
in preface to *Structure and Interpretation of Computer Programs*

Documentation

- Code logic should be easy to read
- Code should be modular
- Code style should be internally consistent
- Code should be documented internally and externally
- Code should be tested and verifiable
- Code should be efficient
- When you write code, think of the person who will have to maintain it

Readable Code

- header comment
- comments at bottom of file show sample run
- in-line comments line up vertically
- vertical whitespace between logical steps
- horizontal whitespace -> opcode -> operands

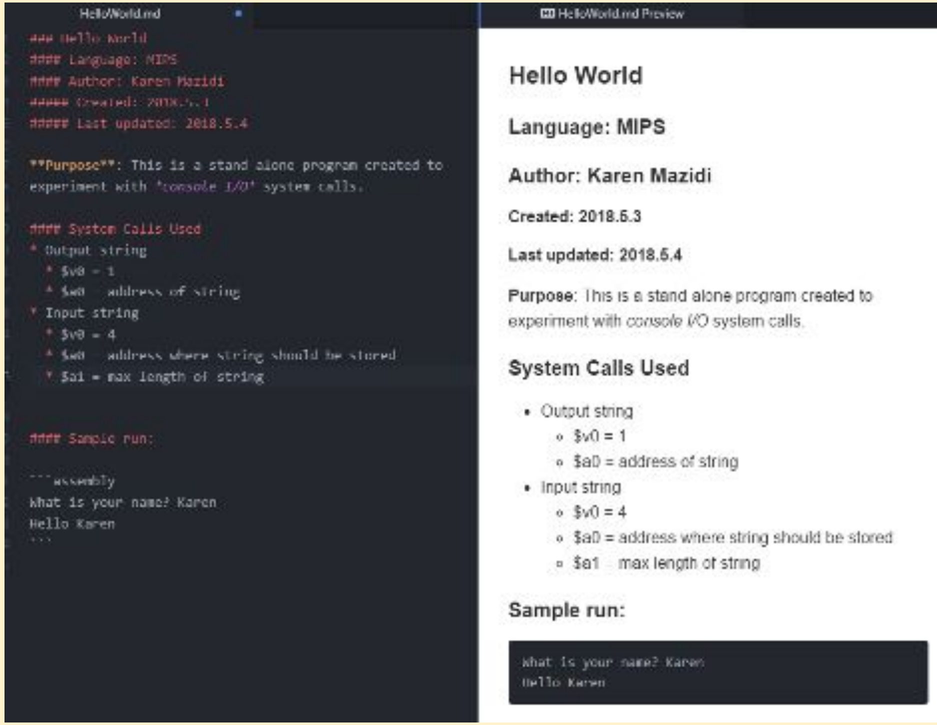


```
##########################################################
# HelloWorld.asm
# author: Karen Mazidi
# created: 2018.5.3
# purpose: demo console I/O
##########################################################

        .data
name:   .space          20
msg1:   .asciiz         "What is your name? "
msg2:   .asciiz         "Hello "

        .text
main:   li      $v0, 4          # print name prompt
        la      $a0, msg1
        syscall

        li      $v0, 8          # read name
        la      $a0, name
        li      $a1, 20
        syscall

        li      $v0, 4          # print hello
        la      $a0, msg2
        syscall

        li      $v0, 4          # print name
        la      $a0, name
        syscall

        li      $v0, 10         # exit
        syscall

# sample run
# What is your name? Karen
# Hello Karen
```

External Documentation

- Atom markdown
- headers start with hashtags
- **bold text**
- *italic text*
- ordered lists with * and indents
- ``` starts and ends code blocks

Markdown cheat sheet:
https://guides.github.com/pdfs/markdown-cheatsheet-online.pdf



~~~
### Hello World
#### Language: MIPS
#### Author: Karen Mazidi
##### Created: 2018.5.3
##### Last updated: 2018.5.4

**Purpose**: This is a stand alone program created to
experiment with *console I/O* system calls.

#### System Calls Used
* Output string
  * $v0 = 1
  * $a0 = address of string
* Input string
  * $v0 = 4
  * $a0 = address where string should be stored
  * $a1 = max length of string


#### Sample run:

```assembly
What is your name? Karen
Hello Karen
```
~~~

Hello World

Language: MIPS

Author: Karen Mazidi

Created: 2018.5.3

Last updated: 2018.5.4

Purpose: This is a stand alone program created to experiment with *console I/O* system calls.

System Calls Used

- Output string
  - $v0 = 1
  - $a0 = address of string
- Input string
  - $v0 = 4
  - $a0 = address where string should be stored
  - $a1 = max length of string

Sample run:

```
What is your name? Karen
Hello Karen
```

GitHub

Why have a GitHub account?

- showcase your coding skills
- keep track of changes for team projects
- it's free (for public repos)

Basics on getting started with GitHub from the command line:

https://github.com/kjmazidi/CS3340/tree/master/GIT_Tutorial