Module 1 – CS 4450

Programming Languages, Paradigms, and Systems

Webber: Chapters 1, 4, and 24

Guiding Questions

- Why are there so many programming languages?
- Why are some more popular than others?
- How can we make informed decisions on which languages to use?
  - What features distinguish the different languages?
  - How would you classify programming languages?

Query

- What languages have **you** used?
- Which do you like best?
- Why?

tiobe.com

- http://www.tiobe.com/tiobe-index/
- Programming popularity/demand index
  - Updated monthly

Aug 2021	Aug 2020	Change	Programming Language	Ratings	Change
1	1		C	12.57%	-4.41%
2	3	↑	Python	11.86%	+2.17%
3	2	↓	Java	10.43%	-4.00%
4	4		C++	7.36%	+0.52%
5	5		C#	5.14%	+0.46%
6	6		Visual Basic	4.67%	+0.01%
7	7		JavaScript	2.95%	+0.07%
8	9	↑	PHP	2.19%	-0.05%
9	14	⇈	Assembly language	2.03%	+0.99%
10	10		SQL	1.47%	+0.02%
11	18	⇈	Groovy	1.36%	+0.59%
12	17	⇈	Classic Visual Basic	1.23%	+0.41%
13	42	⇈	Fortran	1.14%	+0.83%
14	8	⇊	R	1.05%	-1.75%
15	15		Ruby	1.01%	-0.03%
16	12	⇊	Swift	0.98%	-0.44%
17	16	↓	MATLAB	0.98%	+0.11%
18	11	⇊	Go	0.90%	-0.52%
19	36	⇈	Prolog	0.80%	+0.41%
20	13	⇊	Perl	0.78%	-0.33%

Position	Programming Language	Ratings
21	SAS	0.67%
22	Delphi/Object Pascal	0.62%
23	Objective-C	0.61%
24	Rust	0.56%
25	Scratch	0.55%
26	Julia	0.51%
27	Ada	0.49%
28	Lisp	0.47%
29	Dart	0.45%
30	PL/SQL	0.43%
31	(Visual) FoxPro	0.42%
32	Scala	0.42%
33	ABAP	0.41%
34	COBOL	0.37%
35	Logo	0.31%
36	F#	0.30%
37	Kotlin	0.30%
38	Transact-SQL	0.27%
39	Lua	0.26%
40	Scheme	0.24%
41	Ladder Logic	0.24%
42	VBScript	0.23%
43	D	0.23%
44	Clojure	0.22%
45	LabVIEW	0.22%
46	Nim	0.22%
47	VHDL	0.21%
48	Apex	0.21%
49	TypeScript	0.21%
50	Bash	0.18%

Programming Paradigms

- Often grouped into 3 major **families**:
  - Imperative
    - Includes object-oriented
    - Statements, loops, variables
  - Functional
  - Logic

- Some are *multi-paradigm*
  - C++, D, Python, Swift, Rust

Imperative Languages

aka "Procedural"

- Example: a factorial function in C

```
int fact(int n) {
  int sofar = 1;
  while (n > 1)
     sofar *= n--;
  return sofar;
}
```

- Hallmarks of imperative languages:
  - **Assignment**
    - Variables are **fixed** to a memory **location**
    - Updated **in place** through assignment
  - **Iteration**

Functional Languages

- Example: a factorial function in ML

```
fun fact x =
  if x <= 0 then 1 else x * fact(x-1);
```

- Hallmarks of functional languages:
  - *All* functions return a **single value** (no **return** keyword)
    - But could return a list, tuple, etc.
  - **Immutable** variables
    - No **side effects**; no explicit state changes!
    - **No** updating memory in place
  - Heavy use of *recursion* (**not** *iteration*)

Factorial in Other Languages

- C
  - int fact(int x)
    { return x <= 0 ? 1 : x * fact(x-1); }
- Python
  - def fact(n):
    return 1 if x <= 0 else n * fact(n-1)
- Rust (most like ML)
  - fn fact(n: i64) -> i64 {
    if n <= 0 {1} else {n * fact(n-1)}
    }

Strengths and Weaknesses

- The different language groups show their "stuff" on different kinds of problems

- For now, one comment: *don't jump to conclusions based on factorial!*
  - Functional languages do well on such functions
  - Imperative languages, a tiny bit less well
  - Object-oriented languages need *larger examples*
    - They were developed for solving complex problems

About Those Families

- There are *many other* language family terms (not exhaustive and sometimes *overlapping*)
  - Applicative, concurrent, constraint, declarative, definitional, procedural, scripting, …
- Some languages *straddle* families
- Others are so *unique* that assigning them to a family is *pointless*
  - FORTH, APL

Example: Forth Factorial

```
: FACTORIAL
  DUP 1 > IF DUP 1- RECURSE * THEN ;
```

- A *stack-oriented* language
  - Like an HP calculator (X,Y,Z,W stacks)
- Postscript is similar
- Could be called *imperative*, but has little in common with most imperative languages

Example: APL Factorial

$\times / \iota X$

$\times/\iota 5 = \times/1\ 2\ 3\ 4\ 5$
$= 1 \times 2 \times 3 \times 4 \times 5$

- An APL expression that computes X's factorial
- *Expands* X into a vector of the integers 1..X, then *multiplies* them all together
  - (You would not really do it that way in APL, since there is a built-in factorial operator: !X)
- Could be called *functional*, but has little in common with most functional languages

Language Standards

- The documents that define language standards are often drafted by *international committees*
- Can be a slow, complicated and rancorous process
- Fortran: 1982 / 90 / 95 / 2003 / 08 / 15
- C++: 1998 / 2011 / 14 / 17 / 20

- Java and C# evolve faster
  - Why?

The Intriguing Evolution

- Programming languages are evolving rapidly
- New languages are constantly being invented
  - C#, Lua, F#.NET, Scala, Groovy, D, Go, Swift, Rust …
  - F# is ML + objects for .NET
  - Scala is largely an ML front end to the JVM
  - Groovy is a dynamically typed front end to the JVM
  - D and Rust are modern "versions" of C++
  - Go is a modern C
  - Swift is part functional, very cool, but designed for the Mac

Turing Equivalence

- General-purpose languages have different strengths, but fundamentally they all have the *same power*
  - {problems solvable in Java}
    = {problems solvable in Fortran}
    = …
- And all have the same power as various mathematical models of computation
  - = {problems solvable by Turing machine}
    = {problems solvable by lambda calculus}
    = …
- *Church-Turing thesis*: this is what "computability" means

Language Systems

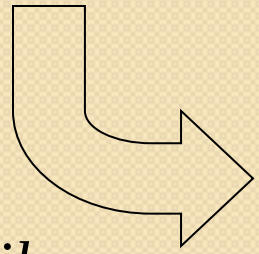
Sequence from source code to executable program

Compiling

- **Compiler** translates to *assembly language*
  - *Machine-specific* instructions
  - C, C++, D, Rust, and Go compile to *native code*
  - Java, C#, Python, Javascript, PHP do **not**

- This is the first step in going from source code to a running program…

```
int i;
int main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```



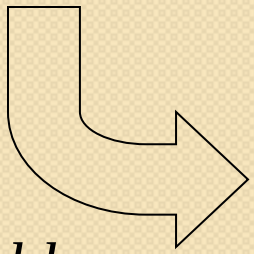
compiler

```
i:      data word 0
main:   move 1 to i
t1:     compare i with 100
        jump to t2 if greater
        push i
        call fred
        add 1 to i
        go to t1
t2:     return
```

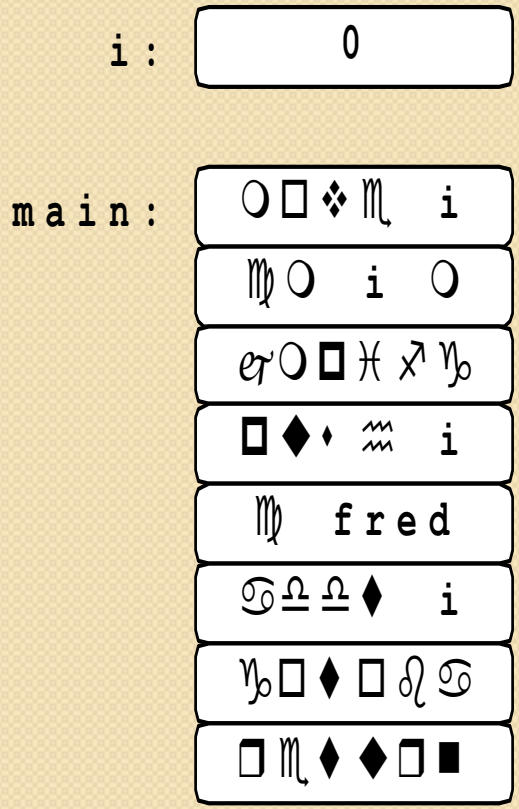
Assembling

- Assembly language is *not* directly executable
  - Still text format, *readable* by (nerdy) people
  - Still has *names*, not memory addresses
- *Assembler* converts each assembly-language instruction into *machine language*
  - Resulting object file not readable by people
- This step is not visible in modern compilers
  - And may even be *skipped*
  - Compilers usually directly yield machine code

```
i:      data word 0
main:   move 1 to i
t1:     compare i with 100
        jump to t2 if greater
        push i
        call fred
        add 1 to i
        go to t1
t2:     return
```

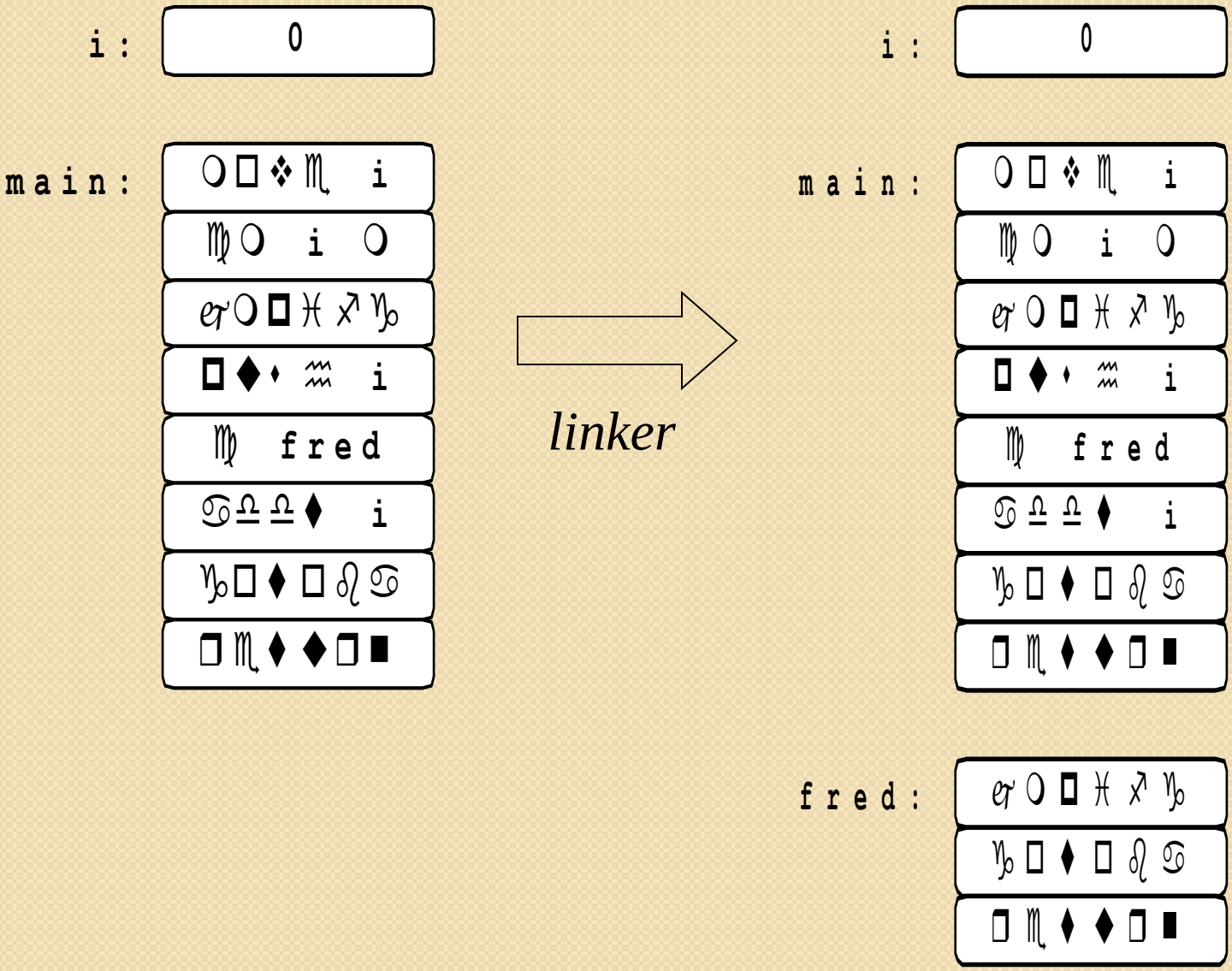


assembler



Linking

- Object file *still* not directly executable
  - Still has some *names*
- Linker combines all the different parts
- In our example, **`fred`** was *compiled separately*,
  - may even have been written in a *different* high-level language
- Result is the *executable file*

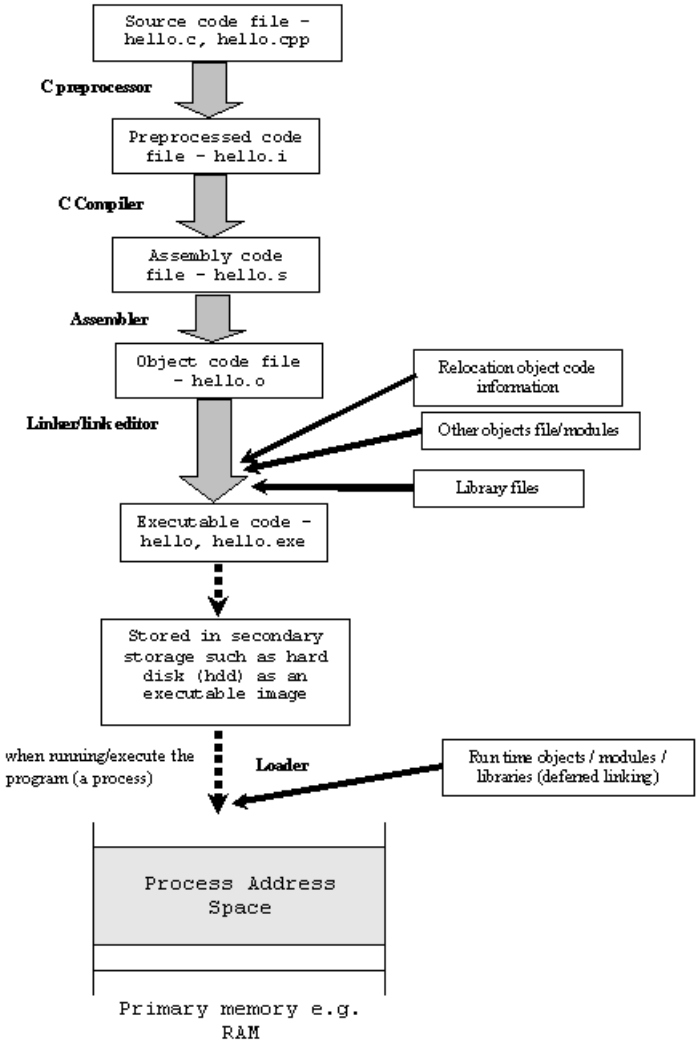

i :
0
main :
fred :
linker

Loading

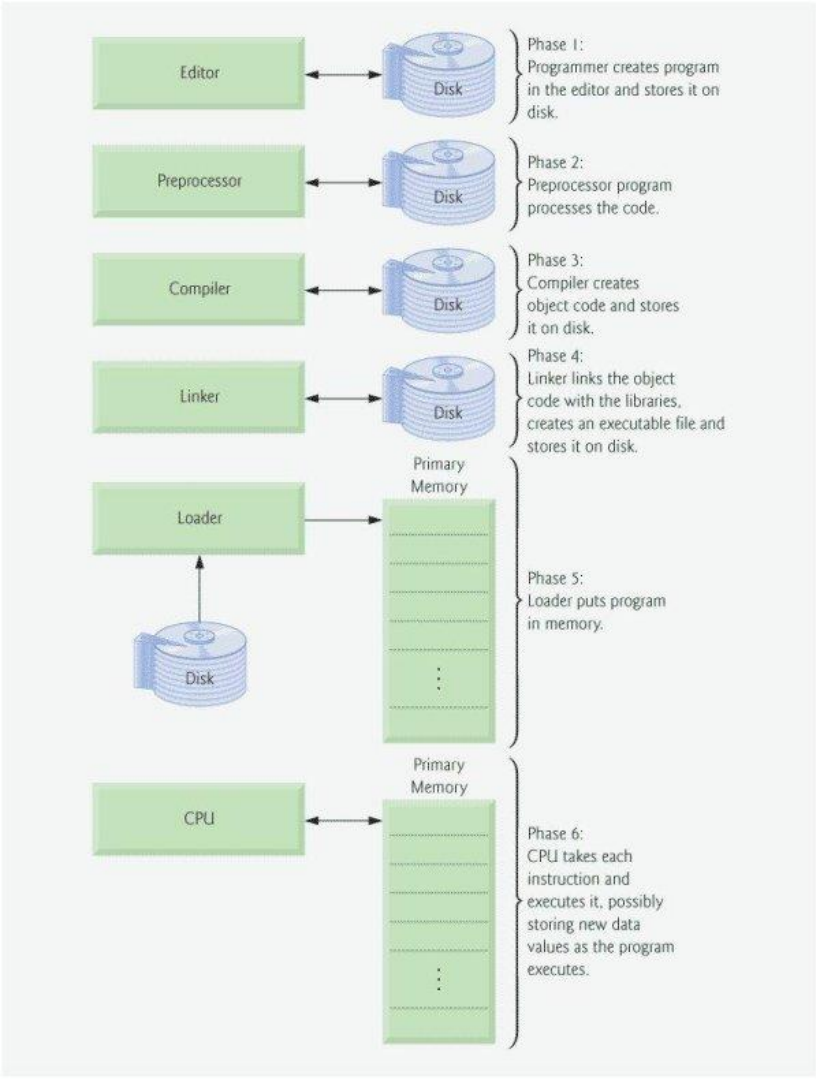
- "Executable" file *still* not directly executable
  - *Still* has some *names*
  - Mostly machine language, but not entirely

- Final step: when the program is run, the *loader* loads it into memory and replaces remaining names with *fixed addresses*

Traditional C/C++ Dev. Process



Modern C/C++ Dev. Process

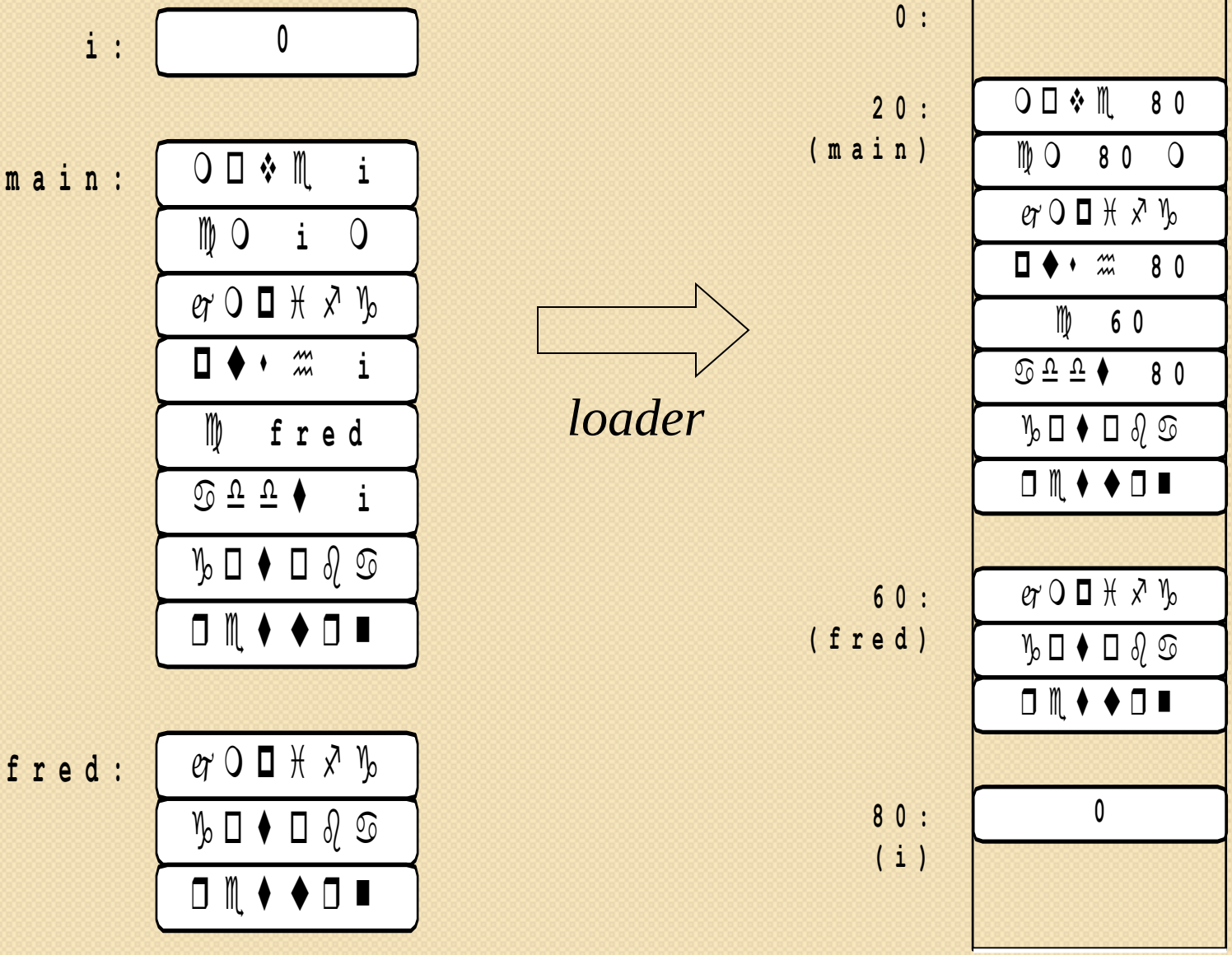


Skips assembler step

A Word About Memory

- Before loading, language system does not know *where* in memory the program will be placed

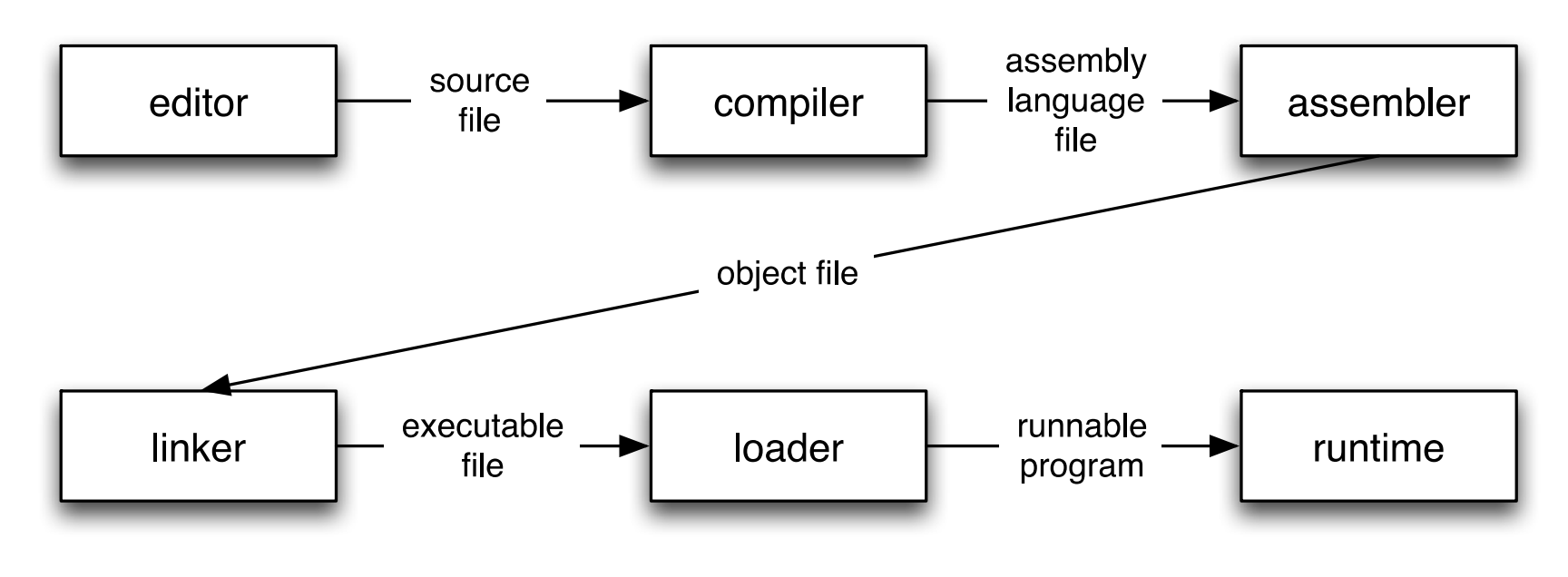
- Loader finds an address for every piece and *replaces names with addresses*
  - like for *static data*
  - There is info in the executable on where the names to be resolved are located
    - By offsets


i :
0
main :
i
i
fred :
0 :
20 :
(main)
80
60 :
(fred)
80 :
(i)
loader

Running

- After loading, the program is *entirely* machine language
  - All names have been replaced with memory addresses
- Processor begins executing its instructions, and the program runs

The Classical Sequence



Languages like Java, C# and Python compile to *byte code*, which the *virtual machine* loads and *interprets*.

About Optimization

- Code generated by a compiler is usually *optimized* to make it faster, smaller, or both
- Other optimizations may be done by the assembler, linker, and/or loader

Example: Loop Invariants

- Original code:

```
int i = 0;
while (i < 100) {
  a[i++] = x*x*x;
}
```

- Improved code, with *loop invariant* moved:

```
int i = 0;
int temp = x*x*x;
while (i < 100) {
  a[i++] = temp;
}
```

The compiler does this (not you!). Unless x is *volatile*.

Example

- *Loop invariant removal* is handled by most compilers
  - That is, most compilers generate the same efficient code as in the previous example
- It is a waste of the programmer's time to make some transformations manually
- "Premature optimization is the root of all evil" (-- Knuth)
  - Premature "pessimization" isn't so great either, though :-)

Other Optimizations

- Some, like LIR, *add variables*
- Others *remove* variables, remove code, add code, move code around, etc.
- All make the connection between source code and object code *more complicated*
- A simple question, such as "What assembly language code was generated for this statement?" may have a complicated answer!
- Very important issue with *concurrency*
  - Instruction reordering at the machine level can result in *race conditions*

Outline

- The classical sequence
- Variations on the classical sequence
- Binding times
- Runtime support

Variation: Hiding The Steps

- Many language systems make it possible to do the compile-assemble-link part with one command
- Example: `gcc` command on a Unix system:

```
gcc main.c
```

Compile-assemble-link

```
gcc main.c –S
as main.s –o main.o
ld …
```

Compile, then assemble, then link

Variation: Integrated Development Environments

- *A single interface* for editing, running and debugging programs
  - Visual Studio, VSCode, XCode, Eclipse, NetBeans, Thonny…

- Integration can add *power* at every step:
  - Editor knows *language syntax*
    - Syntax highlighting, Intellisense (autofill)
  - System may maintain build *versions* (**git**), coordinate collaboration
  - Rebuilding after incremental changes can be coordinated, like Unix **`make`** but language-specific
  - Debuggers can benefit

Variation: Interpreters

- To *interpret* a program is to carry out the steps it specifies, *without first translating* into a lower-level language

- Interpreters are usually much *slower*
  - Compiling takes *more time up front*, but program runs at *hardware speed*
  - Interpreting starts *sooner*, but each step must be processed in *software*

- Sounds like a simple distinction…

Virtual Machines

- A language system can produce code in a "machine language" for which there is no hardware: an *intermediate code*
  - As opposed to assembly language
  - Not machine-specific (hence "virtual" machine)
- A virtual machine must be simulated in software
  - i.e., *interpreted*

Why Virtual Machines

- Cross-platform execution (*portability*)
  - Virtual machine can be implemented in software on many different platforms
- Heightened security
  - *Your* code is *never directly in charge*
    - The *interpreter* is
  - Interpreter can *intervene* if the program tries to do something it shouldn't
    - It runs in a "sandbox"

The Java Virtual Machine

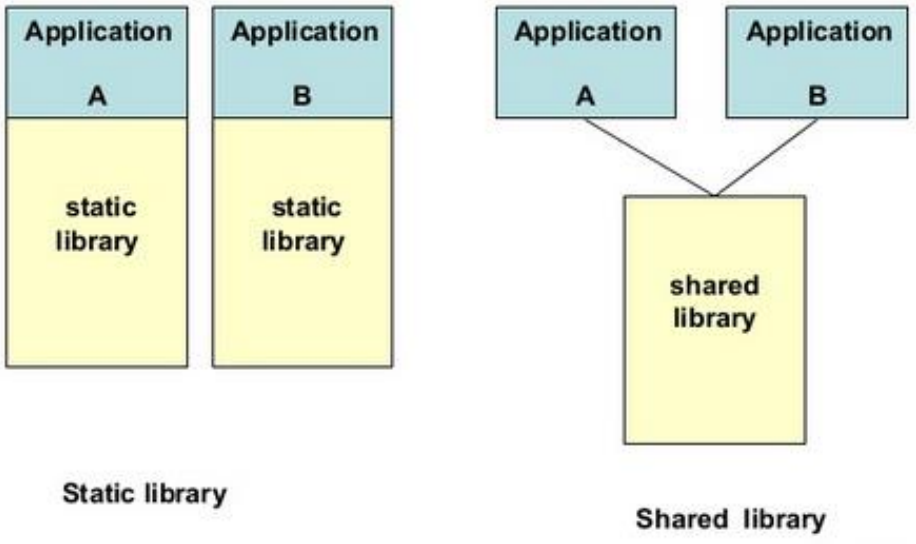
- Java language systems usually compile to code for a virtual machine: the JVM
- JVM language is called *bytecode*
- Bytecode interpreter is part of almost every Web browser
- When you browse a page that contains a Java applet, the browser runs the applet by interpreting its bytecode

Intermediate Language Spectrum

- Pure interpreter (Old BASIC)
  - "Intermediate language" = high-level language
- Tokenizing interpreter (Python)
  - Intermediate language = token stream
- Intermediate-code compiler (Java, C#)
  - Intermediate language = virtual machine language
- Native-code compiler (C++, D)
  - "Intermediate language" = physical machine language

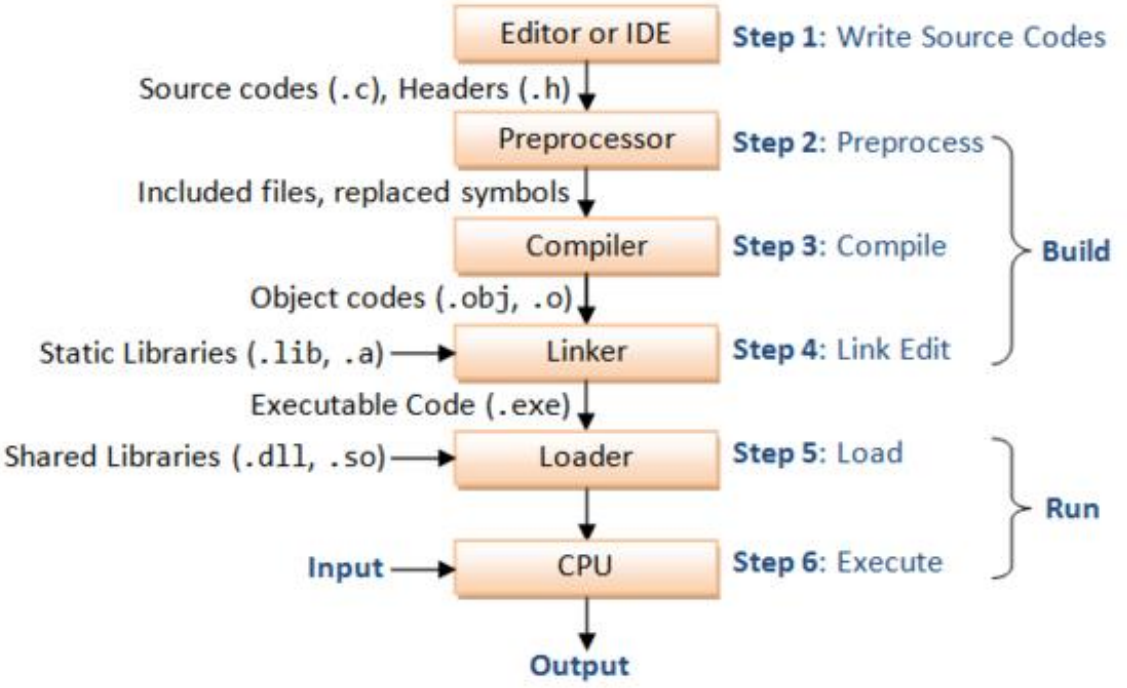
Delayed / Dynamic Linking

- Delays the linking step to *load* or *runtime*
- Code for library functions is *not included* in the executable file for your program



Can you describe 2 advantages of using shared libraries?

Load Time Dynamic Linking



Runtime Dynamic Linking

- You can load shared libraries at **runtime**
- This way you can decide at runtime which version of a library to use
- Each platform provides a system call to load the library on-demand
- Java does this automatically…

Delayed Linking: Java

- JVM automatically loads and links class files from disk when a program uses them
  - $ java myclass
  - At runtime, other classes you reference are automatically loaded

- Java's **class loader** does a *lot* of work:
  - May load across a *network*
  - Thoroughly checks loaded code to make sure it complies with JVM requirements
  - Enforces certain *security* specifications

Delayed Linking Advantages

- Multiple programs can share a copy of library functions: only one copy *on disk*
- Library functions can be *updated* independently of programs: all programs use repaired library code next time they run
- Runtime linking avoids loading code that is never used

Dynamic Compilation

- Some compiling takes place after the program starts running (?!?)
- Many variations:
  - Compile each function only when first called (ML)
  - Start by interpreting, compile only those pieces that are called *frequently*
  - Compile roughly at first (for instance, to intermediate code), then interpret; spend more time on frequently executed pieces (for instance, compile snippets to native code and optimize)
- Just-in-time (JIT) compilation

Outline

- The classical sequence
- Variations on the classical sequence
- Binding times
- Runtime support

Binding

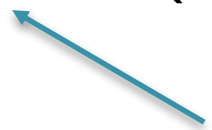
- Binding means *associating* two things—especially, associating some *property* with an *identifier* from the program
- In our example program below:
  - What set of values is associated with **`int`**?
  - What is the type of **`fred`**?
  - What is the address of the object code for **`main`**?
  - What is the value of **`i`**?

```
int i;
int main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

Binding Times

- Different bindings take place at different times

- There is a standard way of describing binding times with reference to the classical sequence:
  - **Language definition time (*standard specification*)**
  - **Language implementation time (*compiler is developed*)**
  - Compile time
  - Link time
  - Load time
  - Runtime



In some languages, the first two binding times above are the same.

Language Definition Time

- Some properties are bound when the language is defined:
  - *Meanings of keywords*: **`void`**, **`for`**, etc.

```
int i;
int main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

Language Implementation Time

- Some properties are bound when the language system is developed for a *specific platform*:
  - *range of values* of type **`int`** in C (but in Java, these are part of the language definition)
  - *implementation limitations*: max identifier length, max number of array dimensions, etc

```
int i;
int main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

Compile Time

- Some properties are bound when the program is compiled or prepared for interpretation:
  - Types of variables, in languages like C and ML that use *static typing*
  - Declaration associated with a given use of a variable, in languages that use *static scoping* (most languages)

```
int i;
int main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

Link Time

- Some properties are bound when separately-compiled program parts are combined into one executable file by the linker:
  - *Pre-compiled machine code* for external functions

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

Load Time

- Some properties are bound when the program is loaded into the computer's memory, but *before* it runs:
  - Exact *Memory locations* for code for functions, static variables
  - Static => "determined *before* runtime"
    - All the binding times *before runtime* are considered to be statically bound

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

Run Time

- Some properties are bound only when the code in question is **executed**:
  - *Values* of variables
  - Location on stack of function parameters and local variables
  - Types of variables, in languages like Lisp and Python that use *dynamic typing*

- Also called *late* or *dynamic* binding
  - Remember: everything before run time is called *early* or *static*
  - Statically bound DLLs on Windows are an anomaly here!

Late Binding, Early Binding

- The most important question about a binding time: late or early?
  - *Late*: generally, this is at runtime; *more flexible* (as with subclass type instances, dynamic loading, etc.)
  - *Early*: generally, this is *faster* and *more secure* than at runtime (less to do at runtime, less that can go wrong)

- You can tell a lot about a language by looking at the binding times

Outline

- The classical sequence
- Variations on the classical sequence
- Binding times
- Runtime support

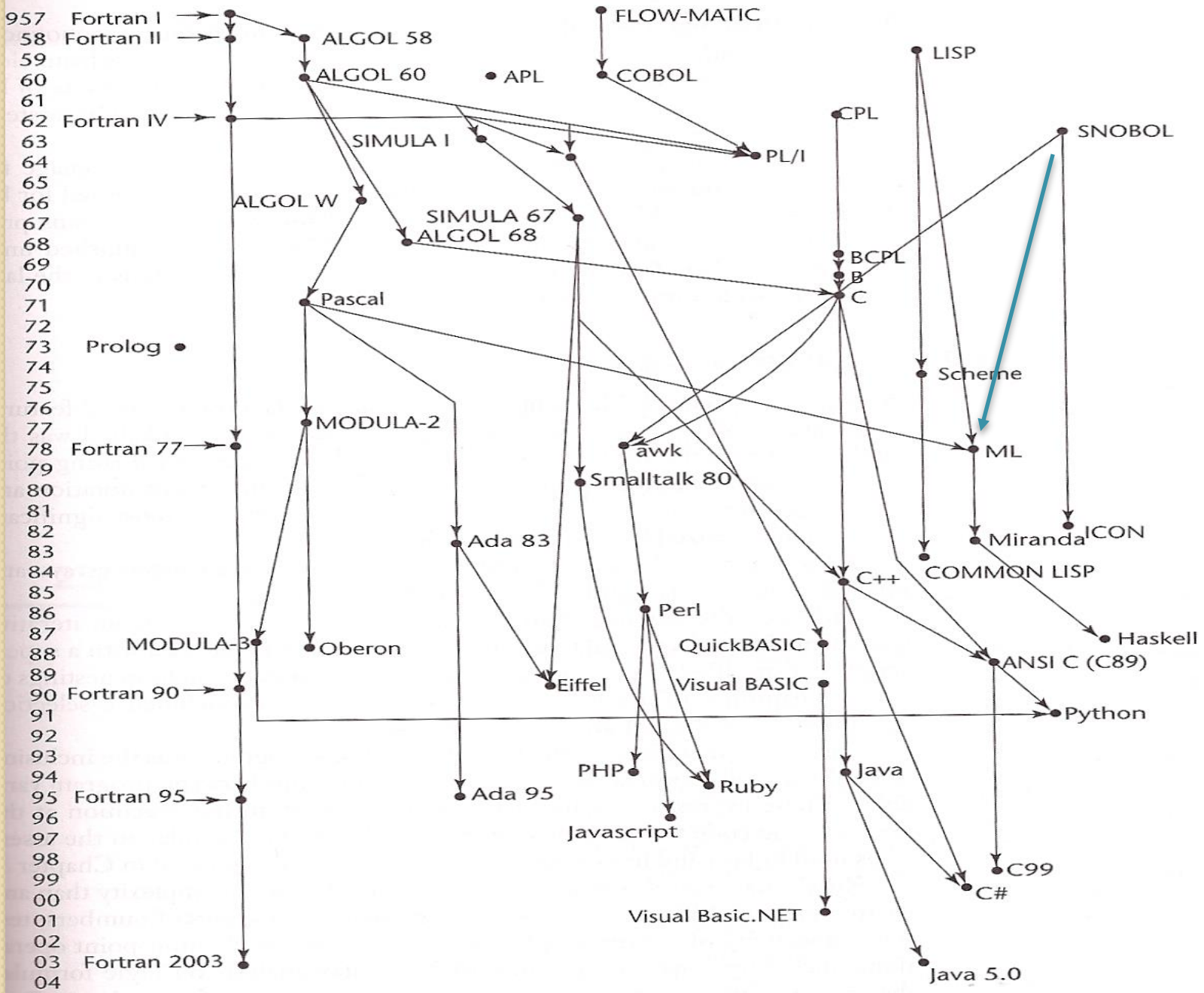
Runtime Support

- Additional code the linker includes even if the program does not refer to it explicitly
  - *Startup processing*: initializing the program state, **argc/argv**
  - *Exception handling*: code that manages exception handlers
  - *Memory management*: allocating and deallocating dynamic memory (stack and heap), garbage collection
  - *Operating system interface*: communicating between running program and operating system for I/O, etc.

- An important *hidden player* in language systems

The History Of Programming Languages

CS 4450 – Chapter 24


957
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
00
01
02
03
04
Fortran I
Fortran II
Fortran IV
Fortran 77
Fortran 90
Fortran 95
Fortran 2003
ALGOL 58
ALGOL 60
ALGOL W
ALGOL 68
FLOW-MATIC
COBOL
APL
SIMULA I
SIMULA 67
PL/I
LISP
CPL
SNOBOL
BCPL
B
C
Pascal
Prolog
MODULA-2
MODULA-3
Oberon
Scheme
ML
awk
Smalltalk 80
Ada 83
Ada 95
Miranda
ICON
COMMON LISP
C++
Haskell
Perl
QuickBASIC
Visual BASIC
Eiffel
ANSI C (C89)
Python
PHP
Ruby
Javascript
Java
C99
C#
Visual Basic.NET
Java 5.0

Fortran

Early 1950s

- The first high-level language
  - The first compiler and linker
- Introduced *scanning*, *parsing*, *register allocation*, *code generation*, and *optimization*
- It is still heavily used in scientific computing
  - Massively parallel processing

Algol
1958–1960

- Introduced:
  - Blocks and static scoping
  - Free format lexical structure
    - Fortran statements started in column 7!
  - Static typing with explicit declarations
  - Nested if-then-else statements
  - Functions are first-class entities (parameters)
  - Operator overloading
- Imperative languages are called "Algol family languages"
  - Most languages are in this family

Lisp
1960

- The first functional programming language
  - Still in use today
- Introduced *conditional expressions*, *recursion*, *dynamic memory* and *garbage collection*
- Influenced modern functional languages like **ML**, **Haskell**, **F#**, **Rust**

ML

1974

- Introduced:
  - Strong static typing
  - Type inference
  - Pattern matching
  - Modules
  - Exceptions
  - Parametric polymorphism (templates/generics)
  - User-defined algebraic types (unions)
- The basis for **Haskell, F#, Scala, Groovy, Rust**

The Rust Language

- From Mozilla
  - Creators of Javascript and Firefox
- A brilliant combination of the best of:
  - C++
  - Python
  - ML
  - Go
  - Lisp
- Faster and safer than C++
  - Only marginally faster, but *much* safer