The Evolution of Major Programming Languages

Different Languages Characteristics

Why have different languages?

- What makes programming languages an interesting subject?
  - The amazing variety
  - The odd controversies
  - The intriguing evolution
  - The connection to programming practice
  - The many other connections

The Amazing Variety

- There are very many, very different languages
- (A list that used to be posted occasionally on **`comp.lang.misc`** had over 2300 published languages in 1995)
- Often grouped into four families:
  - Imperative
  - Functional
  - Logic
  - Object-oriented

Imperative Languages

- Example: a factorial function in C

```
int fact(int n) {
  int sofar = 1;
  while (n>0) sofar *= n--;
  return sofar;
}
```

- Characteristics of imperative languages:
  - Assignment
  - Iteration
  - Order of execution is critical

Functional Languages

- Example: a factorial function in ML

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

- Characteristics of functional languages:
  - Single-valued variables
  - Heavy use of recursion

Another Functional Language

- Example: a factorial function in Lisp

```
(defun fact (x)
  (if (<= x 0) 1 (* x (fact (- x 1)))))
```

- Looks very different from ML
- But ML and Lisp are closely related
  - Single-valued variables: no assignment
  - Heavy use of recursion: no iteration

Logic Languages

- Example: a factorial function in Prolog

```
fact(X,1) :-
  X =:= 1.
fact(X,Fact) :-
  X > 1,
  NewX is X - 1,
  fact(NewX,NF),
  Fact is X * NF.
```

- Characteristics of logic languages
  - Program expressed as rules in formal logic

Object-Oriented Languages

- Example: a Java definition for a kind of object that can store an integer and compute its factorial

```
public class MyInt {
  private int value;
  public MyInt(int value) {
    this.value = value;
  }
  public int getValue() {
    return value;
  }
  public MyInt getFact() {
    return new MyInt(fact(value));
  }
  private int fact(int n) {
    int sofar = 1;
    while (n > 1) sofar *= n--;
    return sofar;
  }
}
```

Object-Oriented Languages

- Characteristics of object-oriented languages:
  - Usually imperative, plus…
  - Constructs to help programmers use "objects"—little bundles of data that know how to do things to themselves

Strengths and Weaknesses

- The different language groups show to advantage on different kinds of problems
- Decide for yourself at the end of the quarter, after experimenting with them

  - Functional languages do well on functions
  - Imperative languages, a bit less well
  - Logic languages, considerably less well
  - Object-oriented languages need larger examples

About Those Families

- There are many other language family terms
  - Concurrent, Declarative, Definitional, Procedural, Scripting, Single-assignment, …
- Some languages straddle families
- Others are so unique that assigning them to a family is pointless

The Odd Controversies

- Programming languages are the subject of many heated debates:
  - User arguments
  - Language standards
  - Fundamental definitions

User Arguments

- There is a lot of argument about the relative merits of different languages
- Every language has users, who praise it in extreme terms and defend it against all others
- To experience some of this, explore newsgroups: **`comp.lang.*`**
- (Plenty of rational discussion there too!)

New Languages

- A clean slate: no need to maintain compatibility with an existing body of code
- But never entirely *new* any more: always using ideas from earlier designs
- Some become widely used, others do not
- Whether widely used or not, they can serve as a source of ideas for the next generation

Widely Used: Java

- Quick rise to popularity since 1995 release
- Java uses many ideas from C++, plus some from Mesa, Modula, and other languages
- C++ uses most of C and extends it with ideas from Simula 67, Ada, ML and Algol 68
- C was derived from B, which was derived from BCPL, which was derived from CPL, which was derived from Algol 60

Not Widely Used: Algol

- One of the earliest languages: Algol 58, Algol 60, Algol 68
- Never widely used
- Introduced many ideas that were used in later languages, including
  - Block structure and scope
  - Recursive functions
  - Parameter passing by value

The Connection To Programming Practice

- Languages influence programming practice
  - A language favors a particular programming style—a particular approach to algorithmic problem-solving
- Programming experience influences language design

Language Influences Programming Practice

- Languages often strongly favor a particular style of programming

  - Object-oriented languages: a style making heavy use of objects
  - Functional languages: a style using many small side-effect-free functions
  - Logic languages: a style using searches in a logically-defined problem space

Fighting the Language

- Languages favor a particular style, but do not force the programmer to follow it

- It is always possible to write in a style not favored by the language

- It is not usually a good idea…

Imperative ML

ML makes it hard to use assignments, but it is still possible:

```
fun fact n =
  let
    val i = ref 1;
    val xn = ref n
  in
    while !xn>1 do (
      i := !i * !xn;
      xn := !xn - 1
    );
    !i
  end;
```

Non-object-oriented Java

Java, more than C++, tries to encourage you to adopt an object-oriented mode. But you can still put your whole program into static methods of a single class:

```
class Fubar {
  public static void main (String[] args) {
    // whole program here!
  }
}
```

Functional Pascal

Any imperative language that supports recursion can be used as a functional language:

```
function ForLoop(Low, High: Integer): Boolean;
  begin
    if Low <= High then
      begin
        {for-loop body here}
        ForLoop := ForLoop(Low+1, High)
      end
    else
      ForLoop := True
  end;
```

Programming Experience Influences Language Design

- Corrections to design problems make future dialects, as already noted

- Programming styles can emerge before there is a language that supports them

  - Programming with objects predates object-oriented languages
  - Automated theorem proving predates logic languages

Other Connections: Computer Architecture

- Language evolution drives and is driven by hardware evolution:
  - Call-stack support – languages with recursion
  - Parallel architectures – parallel languages
  - Internet – Java

Other Connections: Theory of Formal Languages

- Theory of formal languages is a core mathematical area of computer science

  - Regular grammars, lexical structure of programming languages, scanner in a compiler

  - Context-free grammars, parser in a compiler

Language Systems

Language Systems

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

The Classical Sequence

- Integrated development environments are wonderful, but…
- Old-fashioned, un-integrated systems make the steps involved in running a program more clear
- We will look the classical sequence of steps involved in running a program
- (The example is generic: details vary from machine to machine)

Creating

- Remember from last lecture, the programmer uses an editor to create a text file containing the program
- A high-level language: machine independent
- This C-like example program calls **`fred`** 100 times, passing each **`i`** from 1 to 100:

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

Compiling

- Compiler translates to assembly language
- Becomes Machine-specific
- Each line represents either a piece of data, or a single machine-level instruction
- Programs used to be written directly in assembly language, before Fortran (1957)
- Now used directly only when the compiler does not do what you want, which is rare

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

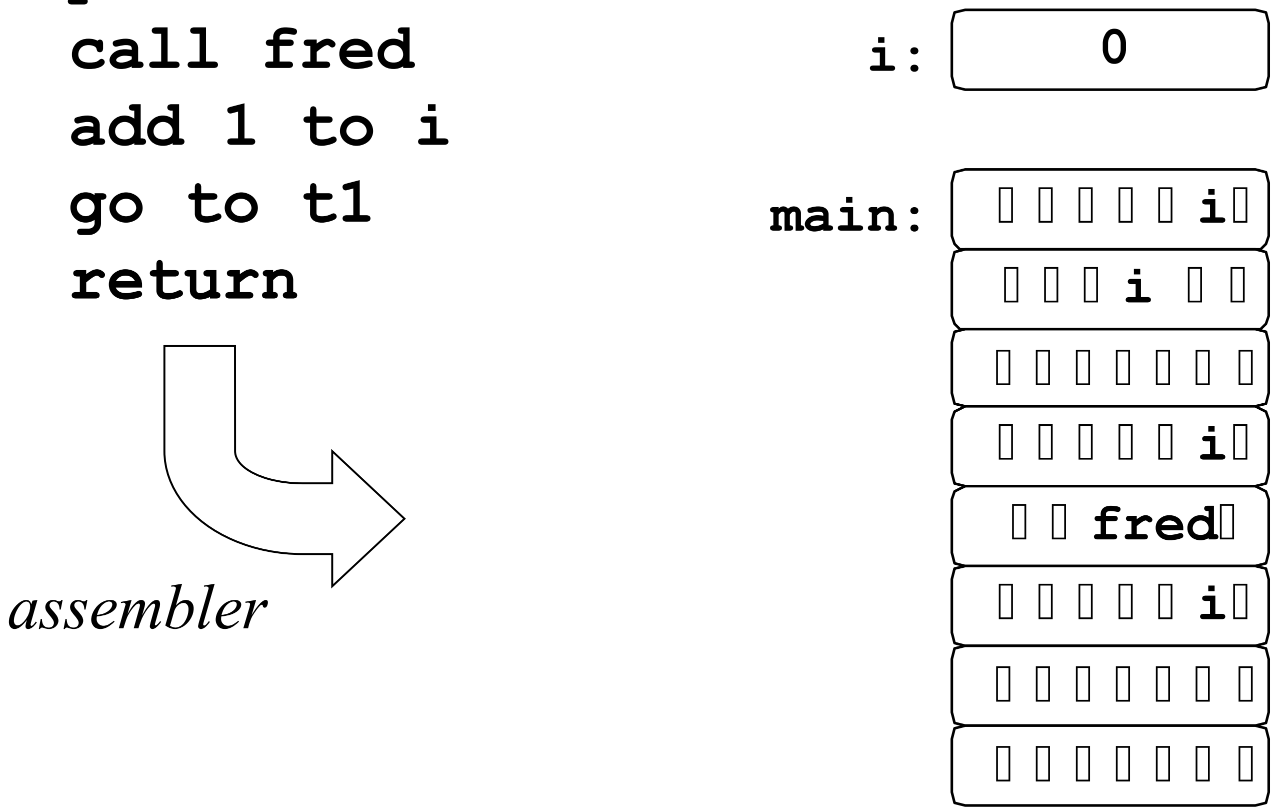
compiler

```
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 still not directly executable
  - Still text format, readable by people
  - Still has names, not memory addresses
- Assembler converts each assembly-language instruction into the machine's binary format: its machine language
- Resulting object file not readable by people

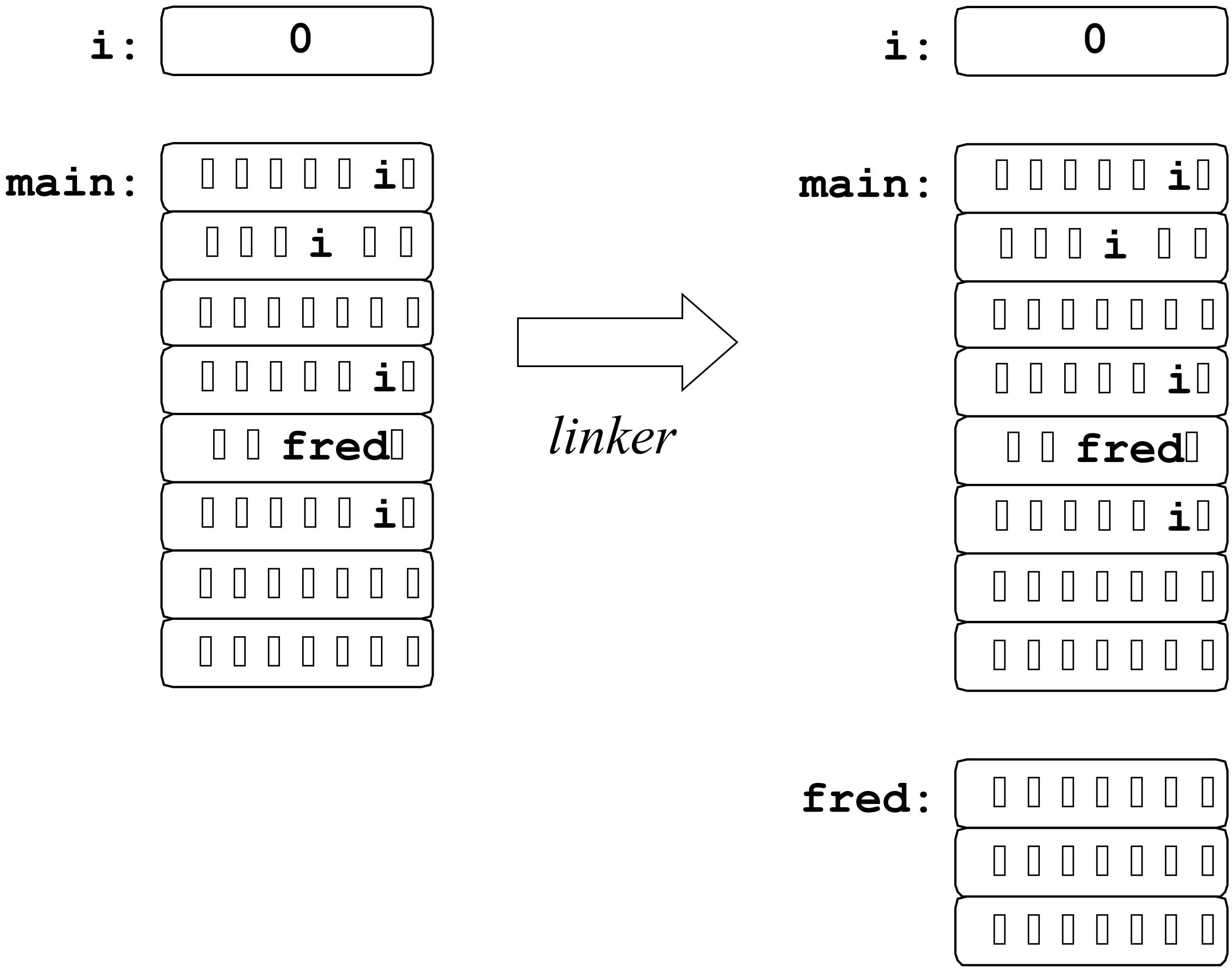
```
i:      data word 0
main:   move 1 to i
t1:     compare i with 100
        jump to t2 if greater
        push i
        call fred
        add 1 to i
        go to t1
t2:     return
```



Linking

- Object file still not directly executable
  - Missing some parts
  - Still has some names
  - Mostly machine language, but not entirely

- Linker collects and combines all the different parts
- In our example, **`fred`** was compiled separately, and may even have been written in a different high-level language
- Result is the executable file


i:
0
main:
i
i
fred
i
linker
i:
0
main:
i
i
fred
i
fred:

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 names with addresses

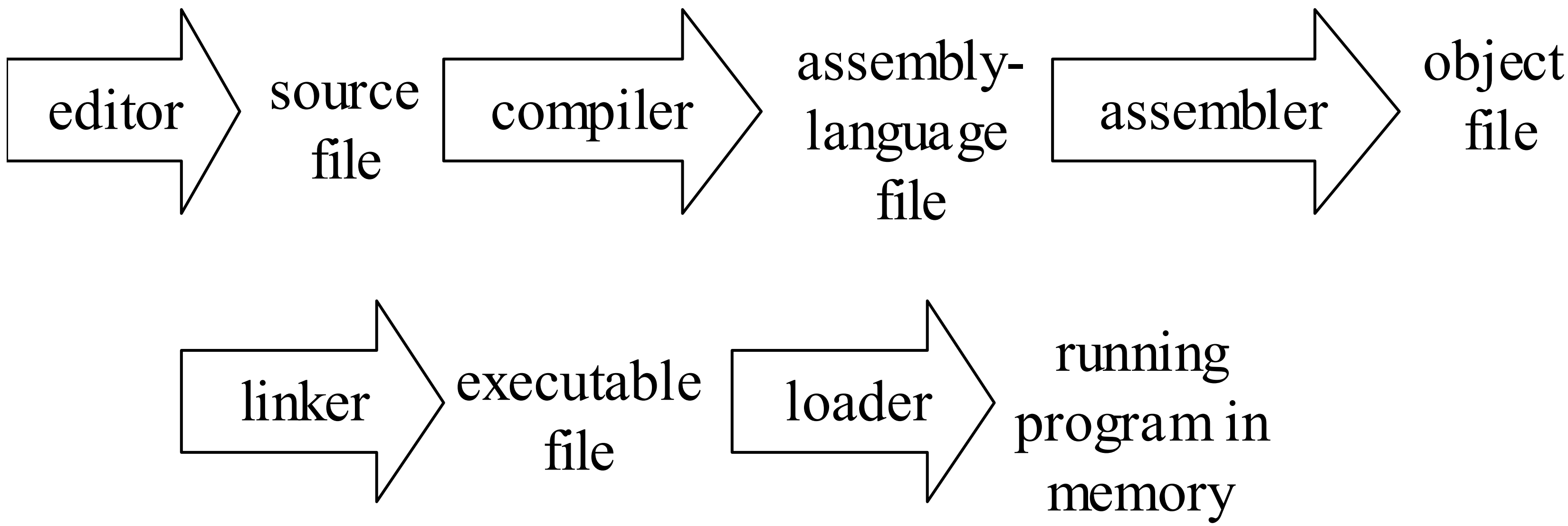
A Word About Memory

- For our example, we are assuming a very simple kind of memory architecture
- Memory organized as an array of bytes
- Index of each byte in this array is its address
- Before loading, language system does not know where in this array the program will be placed
- Loader finds an address for every piece and replaces names with addresses

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



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

Compiling to Object Code

- Many modern compilers incorporate all the functionality of an assembler
- They generate object code directly

Variation: Integrated Development Environments

- A single interface for editing, running and debugging programs

- Integration can add power at every step:
  - Editor knows language syntax
  - System may keep a database of source code and object code
  - System may maintain versions, coordinate collaboration
  - Rebuilding after incremental changes can be coordinated, like Unix `make` but language-specific

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 right away, but each step must be processed in software

Virtual Machines

- A language system can produce code in a machine language for which there is no hardware: an intermediate code
- Virtual machine must be simulated in software – interpreted, in fact
- Language system may do the whole classical sequence, but then interpret the resulting intermediate-code program

Why Virtual Machines

- Cross-platform execution
  - Virtual machine can be implemented in software on many different platforms
  - Simulating physical machines is harder

- Heightened security
  - Running program is never directly in charge
  - Interpreter can intervene if the program tries to do something it shouldn't

The Java Virtual Machine

- Java languages systems usually compile to code for a virtual machine: the JVM
- JVM language is sometimes 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
  - Intermediate language = high-level language
- Tokenizing interpreter
  - Intermediate language = token stream
- Intermediate-code compiler (Java)
  - Intermediate language = virtual machine language
- Native-code compiler
  - Intermediate language = physical machine language

Delayed Linking

- Delay linking step
- Code for library functions is not included in the executable file of the calling program

Delayed Linking: Windows

- Libraries of functions for delayed linking are stored in `.dll` files: dynamic-link library

- Two flavors
  - Load-time dynamic linking
    - Loader finds `.dll` files and links the program to functions it needs, just before running

  - Run-time dynamic linking
    - Running program makes explicit system calls to find `.dll` files and load specific functions

Delayed Linking: Unix

- Libraries of functions for delayed linking are stored in `.so` files: shared object
- Suffix `.so` followed by version number

- Two flavors
  - Shared libraries
    - Loader links the program to functions it needs before running
  - Dynamically loaded libraries
    - Running program makes explicit system calls to find library files and load specific functions

Delayed Linking: Java

- JVM automatically loads and links classes when a program uses them

- Class loader does a lot of work:

  - May load across Internet
  - Thoroughly checks loaded code to make sure it complies with JVM requirements

Delayed Linking Advantages

- Multiple programs can share a copy of library functions: one copy on disk and in memory

- Library functions can be updated independently of programs: all programs use repaired library code next time they run

- Can avoid loading code that is never used

Profiling

- The classical sequence runs twice
- First run of the program collects statistics: For example, parts most frequently executed
- Second compilation uses this information to help generate better code

Dynamic Compilation

- Some compiling takes place after the program starts running

- Many variations:
  - Compile each function only when called
  - Start by interpreting, compile only those pieces that are called frequently
  - Compile roughly at first (for instance, to intermediate code); spend more time on frequently executed pieces (for instance, compile to native code and optimize)
- Just-in-time (JIT) compilation

Binding

- Binding means associating properties with an identifier from the program
  - 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;
void 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
  - Language implementation time
  - Compile time
  - Link time
  - Load time
  - Runtime

Language Definition Time

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

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

Language Implementation Time

- Some properties are bound when the language system is written:
  - range of values of type **`int`** in C
  - implementation limitations: max identifier length, max number of array dimensions, etc

```
int i;
void 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 that goes with a given use of a variable

```
int i;
void 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:
  - Object code for external function names

```
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:
  - Memory locations for code for functions
  - Memory locations for static variables

```
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
  - Types of variables, in languages like Lisp that use dynamic typing
  - Declaration that goes with a given use of a variable (in languages that use dynamic scoping)
- Also called late or dynamic binding

Late Binding, Early Binding

- The most important question about a binding time: late or early?
  - Late: generally, this is more flexible at runtime (as with types & dynamic loading)
  - Early: generally, this is faster and more secure at runtime (less to do, less that can go wrong)
- *You can tell a lot about a language by looking at the binding times*

Debugging Features

- Examine a snapshot, such as a core dump
- Examine a running program on the fly
  - Single stepping, break-pointing, modifying variables
- Modify currently running program
  - Recompile, re-link, reload parts while program runs

- *Advanced debugging features require an integrated development environment*

Debugging Information

- Where is it executing?
- What is the traceback of calls leading there?
- What are the values of variables?
- Source-level information from machine-level code
  - Variables and functions by name
  - Code locations by source position
- Connection between levels can be hard to maintain, for example because of optimization

Runtime Support

- Additional code the linker includes even if the program does not refer to it explicitly
  - Startup processing: initializing the machine state
  - Exception handling: reacting to exceptions
  - Memory management: allocating memory, reusing it when the program is finished with it
  - Operating system interface: communicating between running program and operating system for I/O, etc.

The End

- Read Chapter 3 – Describing Syntax and Semantics