Introduction to Compiler Design

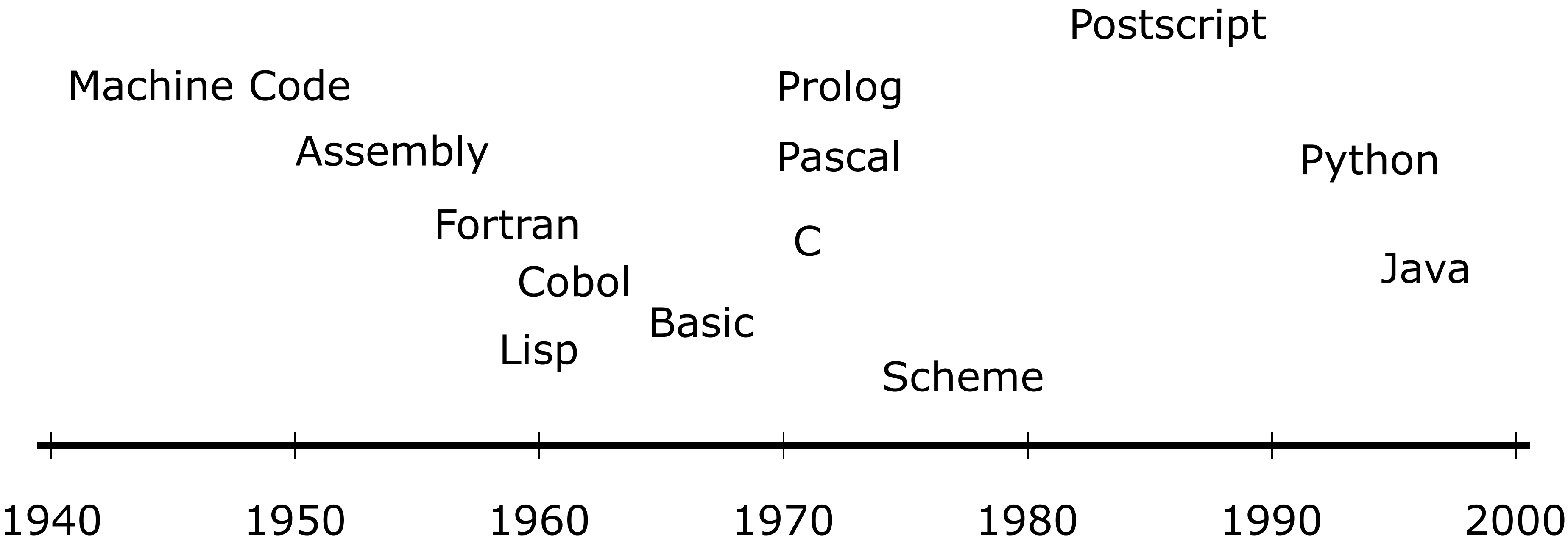
Lesson 2:

Programming Language Basics

The Make utility

Programming language basics

Evolution of Programming Languages



Machine Code

Assembly

Fortran

Lisp

Cobol

Basic

Prolog

Pascal

C

Scheme

Postscript

Python

Java

1940 1950 1960 1970 1980 1990 2000

Types of Programming Languages

- Imperative Languages

  Languages which specify HOW a computation is to be done.

  C, C++, C#, Java, Python, Perl, …

- Declarative Languages

  Languages which specify WHAT computation is to be done.

  ML, Prolog, Haskell, …

Programming Language Basics

- Static/Dynamic Distinction
- Environments and States
- Static Scope and Block Structure
- Explicit Access Control
- Dynamic Scope
- Parameter Passing
- Aliasing

Static / Dynamic Distinction

- Static

  Issue can be decided at compile time

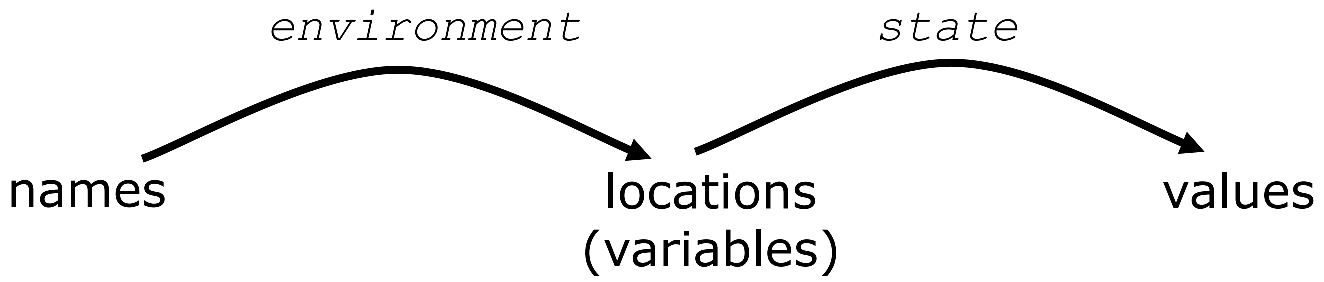
- Dynamic

  Issue cannot be decided until runtime

Example:

public ~~static~~ int x;

Environments and States



- Static vs. Dynamic binding of names to locations

  Globals can be static, others dynamic

- Static vs. Dynamic binding of locations to values

  Constants can be static, others dynamic (Strings in Java are imutable)

Static Scope and Block Structure

```
main() {
    int a = 1;
    int b = 1;
    {
        int b = 2;
        {
            int a = 3;
            cout << a << b;
        }
        {
            int b = 4;
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

Block

Declaration D “belongs” to block B
If B is the most closely nested
block containing D.

Scope of declaration D is the block
Containing D and all sub-blocks
That don’t redeclare D.

Explicit Access Control

- Classes introduce new scoping for data members.
- Subclasses act like sub-blocks
- public, private, and protected limit access to data members

Dynamic Scope

Use of name x refers to the declaration of x in the most recently called, not-yet-terminated, procedure with such a declaration

```
class Foo {
        public void x(){
        }
}

class Bar extends Foo {
        public void x(){
        }
}
```

```
…
Foo foo;
…
foo.x();
…
```

Which version of x() is called?

Dynamic Scoping vs. Static Scoping

- Static is most closely related declaration in space
- Dynamic is most closely related declaration in time

Parameter Passing

How do *actual parameters* associate to *formal parameters*?

- Call by Value

  A copy of actual parameter is made
  and placed in formal parameter

- Call by Reference

  The address of actual parameter is passed
  as value of the formal parameter

Aliasing

- When two names refer to the same location in memory

- Affects optimization step of compilers

The Make utility

Makefiles: Motivation

- Typing the series of commands to generate our code can be tedious
  - Multiple steps that depend on each other
  - Somewhat complicated commands
  - May not need to rebuild everything
- Makefiles solve these issues
  - Record a series of commands in a script-like DSL
  - Specify dependency rules and Make generates the results

Makefiles: Basic Structure

```
<target>:  <dependency list>
  (tab)<command to satisfy target>
```

Makefiles: Basic Structure

<target>: <dependency list>

(tab)<command to satisfy target>

Example

```
Example.class: Example.java IO.class
	javac Example.java

IO.class: IO.java
	javac IO.java
```

Makefiles: Basic Structure

<target>: <dependency list>

(tab)<command to satisfy target>

Example

```
Example.class: Example.java IO.class
        javac Example.java

IO.class: IO.java
        javac IO.java
```

Example.class depends on example.java and IO.class

Makefiles: Basic Structure

<target>: <dependency list>

  (tab)<command to satisfy target>

Example

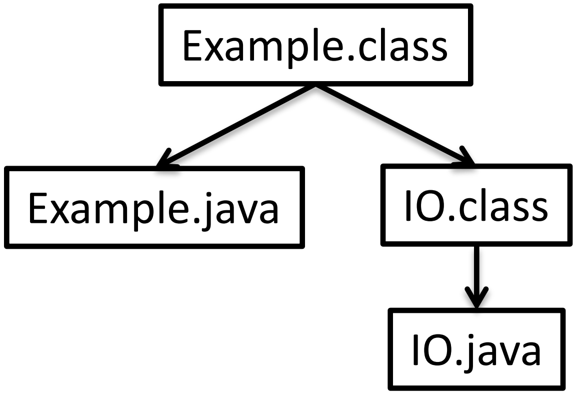
```
Example.class: Example.java IO.class
    javac Example.java

IO.class: IO.java
    javac IO.java
```

Example.class is generated by javac Example.java

Example.class depends on example.java and IO.class

Makefiles: Dependencies



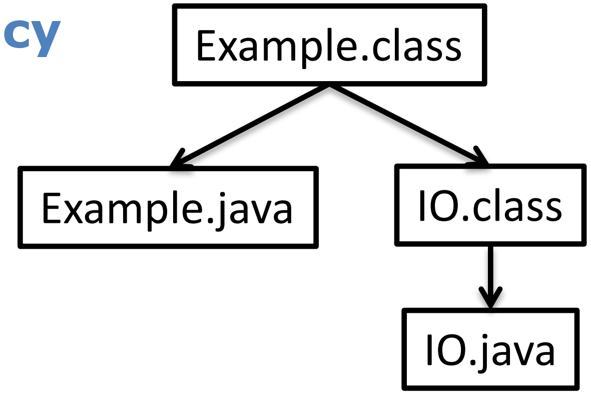
Example

```
Example.class: Example.java IO.class
	javac Example.java

IO.class: IO.java
	javac IO.java
```

Makefiles: Dependencies

Internal Dependency graph



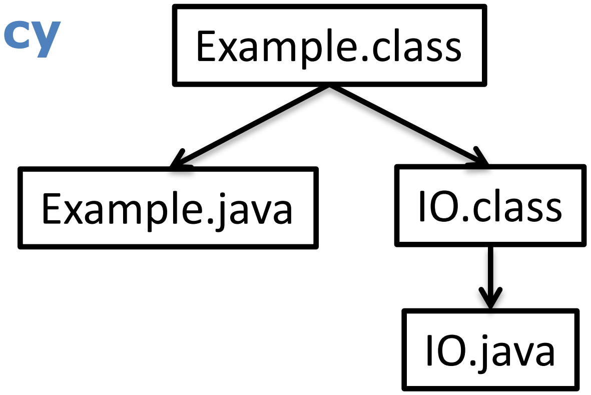
Example

```
Example.class: Example.java IO.class
	javac Example.java

IO.class: IO.java
	javac IO.java
```

Makefiles: Dependencies

Internal Dependency graph



Example.class

Example.java

IO.class

IO.java

A file is rebuilt if one of its dependencies changes

Example

```
Example.class: Example.java IO.class
	javac Example.java

IO.class: IO.java
	javac IO.java
```

Makefiles: Variables

You can thread common configuration values through your makefile

Makefiles: Variables

You can thread common configuration values through your makefile

Example

```
JC = /s/std/bin/javac
JFLAGS = -g
```

Makefiles: Variables

You can thread common configuration values through your makefile

Example

JC = /s/std/bin/javac

JFLAGS = -g **Build for debug**

Makefiles: Variables

You can thread common configuration values through your makefile

Example

JC = /s/std/bin/javac

JFLAGS = -g **Build for debug**

```
Example.class: Example.java IO.class
    $(JC) $(JFLAGS) Example.java

IO.class: IO.java
    $(JC) $(JFLAGS) IO.java
```

Makefiles: Phony Targets

- You can run commands via make
  - Write a target with no dependencies (called phony)
  - Will cause it to execute the command every time

Makefiles: Phony Targets

- You can run commands via make
  - Write a target with no dependencies (called phony)
  - Will cause it to execute the command every time

Example

```
clean:
      rm –f *.class
```

Makefiles: Phony Targets

- You can run commands via make
  - Write a target with no dependencies (called phony)
  - Will cause it to execute the command every time

Example

```
clean:
     rm –f *.class
test:
     java –cp . Test.class
```

Running Make

- Type

```
make target-name
```

- Or just type

```
make
```

The *first* target will be created

- Try it out (login to linux machine)

More with Make

```
test: examples.class
  java examples $(INPUT)
```

then type the command:

```
make test INPUT=in.data
```

More About Make

For a complete description:

https://www.gnu.org/software/make/manual/make.html

For a short introductory tutorial:

make-tutorial.pdf (online on the web page)