

Programming Languages

The birds and animals are all friendly to each other, and there are no disputes about anything. They all talk, and they all talk to me, but it must be a foreign language for I cannot make out a word they say.
-- Mark Twain

■ Why?



1.1

- ▲ computers, programming, abstraction
- ▲ programming languages
- ▲ courses about programming languages

■ Background



1.2, 1.5, 2.*

- ▲ classes of languages
- ▲ history

■ Housekeeping

1

Computers, Programming

The computer is a very *general purpose* machine, capable of any number of *special purpose* tasks.

■ special purpose machine

- ▲ a small number of *complex* operations
- ▲ user as director of complex operations

■ general purpose machine

- ▲ a small number of very *simple* operations
- ▲ programmer as architect, building complex operations by combining simple operations in meaningful ways
- ▲ user as director of complex operations
- ▲ can be used to simulate many different special purpose machines



Examples of special purpose machines? General purpose?

2

Example

Let's say we have a very simple general purpose machine with the following basic operations:

- `move(Const, X)` *store Const in X*
- `incr(X)` *increment the value in X*
- `sign(X)` *change the sign of the value in X*
- `rept(Op, X)` *repeat Op operation X times*

We can invent new operations by combining operations:

- `rept(incr(Y), X)`
- `sign(X); incr(X); sign(X)`

3

Abstraction

By giving our new operations *names* we can use them without ever having to consider the details of their implementation:

- `add(X, Y)` `rept(incr(Y), X)`
- `decr(X)` `sign(X); incr(X); sign(X)`

We now have a slightly more powerful machine with more operations.

Abstraction is the reduction of a complex object or process to a simple form that can be used without considering the inner complexities.

4

Examples of Abstraction

- `add(X, Y)`

- `Y := Y + X;`

- `x := sqrt(y);`

- `today = (99,12,31);`
`today++;`

- *"the queen is pregnant"*



- `:^7`

5

Programming Languages

- A computer has a small set of primitive operations and a small set of primitive data types.
- A *Programming Language* is a set of more complex operations and data types along with rules for combining them. The operations and types are given names *so that they can be used without regard for the complexities of their implementation*.
- That is, programming languages provide *abstractions* of combinations of operations and data.
 - ▲ Different languages approach abstraction in different ways, with different emphases.
 - ▲ Programming languages also allow the programmer to create procedural and data abstractions (to varying degrees)

6

Programming Language Courses

Learning a programming language is like learning to drive.
Learning a new language is like learning to drive something else.*

- sometimes it's like learning to drive a standard
- sometimes it's like learning to drive a boat
- sometimes it's like learning to ride a motorcycle

Possibly the most important reason to study programming languages is to learn that programming goes way beyond the superficial syntax of any particular language.

*(Learning assembly language is like learning to *build* a car).

7

Programming Language Courses (cont.)

What's the point of studying *programming languages* independently of any particular language?

- Expressiveness
 - ▲ built-in features in one language can inspire implementation ideas in a language without those features
 - ▲

```
char myarray[16], upperarray[16];  
...  
map(uppercase, 1, myarray, upperarray);  
...  
uppercase(char c)  
{  
    return((c > 96 && c < 123) ? c - 32 : c);  
}  
  
map(void *funcptr(), int arity, ...
```

8

Programming Language Courses (cont.)

■ The “Right” Language

- ▲ things that are difficult to program in one language may be simple to program in a different language

```
▲ qsort [] = []
   qsort (x:xs) = qsort smalls ++ [x] ++ qsort bigs
                   where
                       smalls = [y | y <- xs, y < x]
                       bigs   = [y | y <- xs, y >= x]
```

- ▲ things that are inefficient in one language may be efficient in a different language

 *Advantages/disadvantages of writing quicksort in Haskell?*

9

Programming Language Courses (cont.)

■ The Learning Curve

- ▲ learning many languages *and things common to all languages* makes learning new languages easier

```
▲ OUTER:for i in 1 .. MAX loop
    j := 1;
    INNER:loop
        prod := prod * (i + j);
        exit OUTER when prod > 2**12;
        j := j + i;
        exit INNER when j > 10;
    end loop INNER;
end loop OUTER;
```

10

Programming Language Courses (cont.)


■ Programming Insight

- ▲ knowing how languages are implemented gives a better understanding when using them

▲ ...
dothis();
...
}

dothis()
{
 register tmpvar = 0L;
 ...
}

may crash here!



11

Programming Language Courses (cont.)

■ Language Description

- ▲ learning how to *describe* the form and meaning of programming languages allows you to:
 - learn new languages quickly and fully
 - understand ambiguity in languages
 - evaluate languages
 - write parsers for languages
 - use language tools (*e.g.*, compiler compilers)
 - develop new languages
 - command languages, scripting languages, full-blown programming languages
 - *prove* properties of programs

12

Classes of Programming Languages

There are different ways of grouping programming languages together based on their similarities:

- by domain
 - ▲ business languages, scientific languages, AI languages, systems languages, scripting languages
- by generality
 - ▲ general purpose vs. special purpose
- by paradigm
 - ▲ a *paradigm* is a way of viewing programming, based on underlying theories of problem solving styles
 - ▲ programming languages grouped in the same paradigm are similar in their approach to problem solving
 - ▲ imperative, object-oriented, logic-based, functional

13

Classifying Languages by Domain: Business

Historically, languages were classified most often by domain.

- Business (sometimes a.k.a. *data processing*)
 - ▲ language features emphasize file handling, table lookup, report generation, etc. (*high level of abstraction*)
 - ▲ weak language support for math functions, graphics, recursion, etc. (*low level of abstraction*)
 - ▲ Cobol
 - ▲ now mostly handled by database systems, spreadsheets, etc. (*which often include their own special purpose languages, by the way*)

 *Examples of business applications?*

14

Classifying Languages by Domain: Scientific

■ Scientific

- ▲ language features emphasize math functions, graphics, recursion, concurrency, programmer defined abstraction, etc.
- ▲ weak language support for file manipulation, table lookup, report generation, etc.
- ▲ often low-level, efficient
- ▲ Fortran, Algol, Pascal, etc.

 *Examples of scientific applications?*

15

Classifying Languages by Domain: AI

■ Artificial Intelligence

- ▲ high level of abstraction for symbol manipulation, linked lists (often built-in), recursion, self-modification, etc.
- ▲ basic support only for file manipulation, graphics, user interface, etc.
- ▲ often high-level, inefficient
- ▲ Lisp, Prolog, Scheme, ML, etc.

 *Examples of AI applications?*

16

Classifying Languages by Domain: Systems

■ Systems

- ▲ language support for hardware interface, operating system calls, direct memory/device access, etc.
- ▲ little or no direct support for programmer defined abstraction, complex types, symbol manipulation
- ▲ very low-level, very efficient
- ▲ very few restrictions on programmer (access to everything)
- ▲ macro assembler, C

 *Examples of systems applications?*

17

Classifying Languages by Domain: Scripting

■ Scripting

- ▲ language support for system calls (to execute commands, programs, etc.), string manipulation, file handling
- ▲ basic support only for numeric types, loops, user interface, etc.
- ▲ csh, .bat, awk, Perl, etc.

 *Examples of scripting applications?*

18

Classifying Languages by Generality

■ General Purpose

- ▲ languages with features that allow implementation of virtually any algorithm
- ▲ roughly uniform level of abstraction over language features
- ▲ Pascal, C, C++, Java, Lisp, Prolog, Delphi, etc., etc., etc.

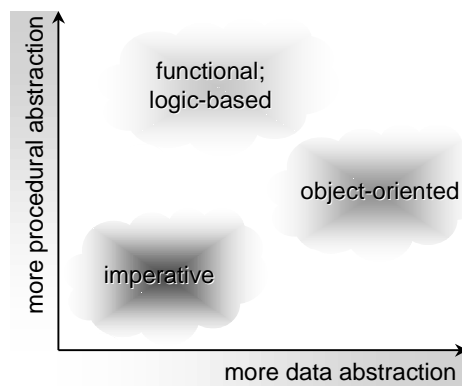
■ Special Purpose

- ▲ languages with a very restricted set of features
- ▲ high level of abstraction among features
- ▲ SQL, WordBasic, etc.

19

Classifying Languages by Paradigm

Now probably the most common way of classifying languages.



20

Classifying Languages by Paradigm: Imperative

■ Imperative Languages

- ▲ a computer functions by executing simple instructions one after another
- ▲ imperative languages mirror this behaviour at a slightly higher level of abstraction
 - the programmer specifies operations to be executed and specifies the order of execution to solve a problem
 - the imperative language operations are themselves just abstractions of some sequence of lower-level machine instructions
- ▲ programmer must still describe in detail *how* a problem is to be solved (*i.e.*, all of the steps involved in solving the problem)
- ▲ Assembly, Fortran, Algol, Ada, Pascal, C, etc.

21

Classifying Languages by Paradigm: Object-Oriented

■ Object-Oriented Languages

- ▲ a problem is solved by specifying *objects* involved in the problem
 - objects in OO correspond roughly to real-world objects in the problem
 - objects are instances of *classes* (abstract data types)
 - classes are arranged in *hierarchies*, with subclasses inheriting properties of superclasses
 - operations (called methods) are defined specific to each class
 - problem solving is accomplished through *message passing* between objects (a *message* is a call to a method of a specific object)
- ▲ Smalltalk, Dylan, Java, etc.
- ▲ hybrids: Delphi, Ada95, C++, Prolog++, CLOS, LIFE, etc.

22

Classifying Languages by Paradigm: Logic-Based

■ Logic-Based Languages

- ▲ a problem is solved by stating the problem in terms of logic (usually first-order logic, a.k.a. predicate calculus)
 - a program consists of known facts of the problem state as well as rules for combining facts (and possibly other rules)
 - program execution consists of constructing a *resolution proof* of a stated proposition (called the *goal*)
 - the entire *theorem prover* is built-in to the language and is not visible to the programmer (*major procedural abstraction!*)
- ▲ Prolog, Goedel, Mercury

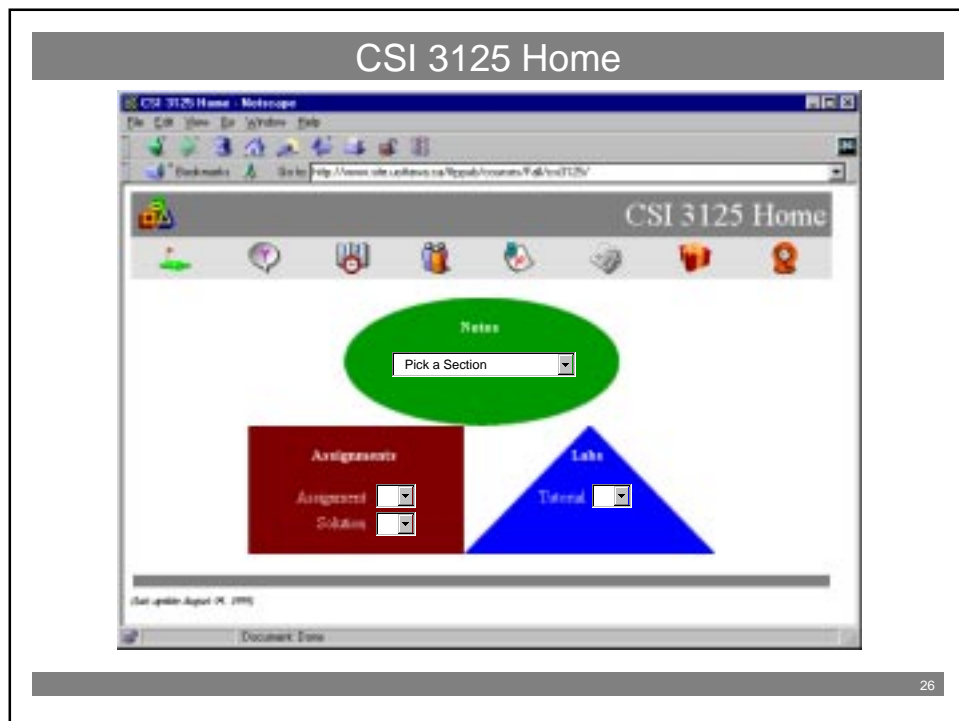
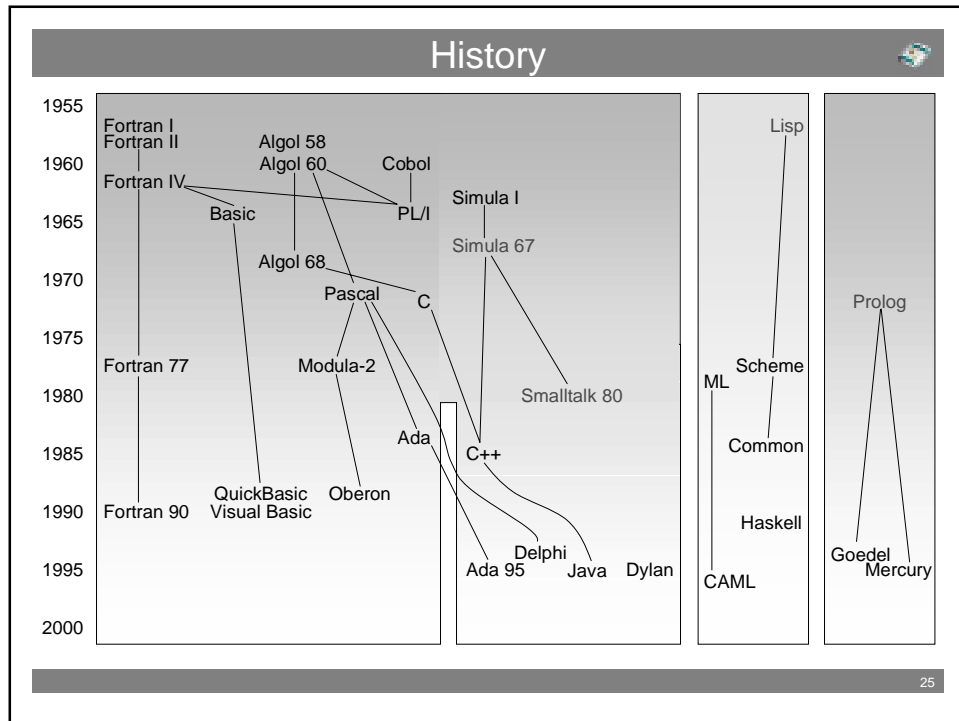
23

Classifying Languages by Paradigm: Functional

■ Functional Languages

- ▲ a problem is solved as the evaluation of a function
 - a program consists of a set of function definitions, where a function is simply a mapping from elements of one set to elements of another set
 - program execution consists of evaluating a top-level function (*i.e.*, applying a function to specified elements of a set)
 - the language itself is the function evaluator; the evaluation mechanism is not visible to the programmer (*major procedural abstraction!*)
 - no variables, no assignment
 - “everything is a function”
 - can apply functions to functions too!
- ▲ Lisp, Scheme, Common Lisp, ML, CAML, Haskell

24



Marks

<i>Assignments (A)</i>	30 marks
<i>Midterm exam (M)</i>	25 marks
<i>Final exam (F)</i>	45 marks

Grade (G) 100 marks

*As usual, if you get a failing mark on the exams ($< 35/70$),
you get a failing grade in the course
($28 \leq E < 35$; $F < 28$)*

27

Tutorials

DGD1

Wed 16:00-18:00

DGD2

Wed 16:00-18:00

DGD3

Tue 08:30-10:30

28