Language Design, Economics of Programming Languages, and Overview of COOL

Grade Weights

- Project 50%
  - I, II 10% each
  - III, IV 15% each
- Midterm 15%
- Final 25%
- Written Assignments 10%
  - 2.5% each

Lecture Outline

- Today's topic: language design
- Why are there new languages? (The Economy of Languages)
- Good-language criteria
- History of ideas:
  - Abstraction
  - Types
  - Reuse
- Cool
- The Course Project

Programming Language Economics: Three Questions

- Why are there so many programming languages?
  - Hundreds if not thousands in everyday use.
  - Why do we need these?
  - Why not just one all-purpose language?
- Why are there new programming languages given that we already have so many?
- How do we know a good programming language when we see one?

Why So Many Programming Languages?

- Well, why so many different models of cars?
  - Fast cars
  - Cars with good fuel economy
  - Cars (i.e., trucks, SUVs) that can carry larger amounts of cargo
  - Cars for various terrains
- Pretty clear why one model of car would not suffice for all potential uses: No one car can do all things for all drivers.

Programming Domains

- Scientific Computing (big science, long running apps, simulations)
  - Good floating point support
  - Good support for arrays and operations on arrays of floating point data
  - Good support for parallelism
  - Many other required qualities
- Solution: (modern) FORTRAN
  - still used by many scientists today

Programming Domains

- » Business Computing
  - » Persistence: don’t want to lose your data
  - » Good facilities for report generation
  - » Good support for data analysis
- » Solution: SQL and related relational database programming languages

Programming Domains

- » Systems programming (e.g., embedded systems, control systems, operating systems)
  - » Very fine grained control of low level resources
  - » Support for handling time constraints
    - » e.g., network response times
- » Solution: C, C++

Programming Domains

- Mobile computing
  - Ease of network functionality
  - Appropriate mechanisms for inter-process communication
  - Capability to provide applications for variety of mobile platforms
    - Differing power and memory capacities, etc.
- Solution?: Android versions of Java?

Why Are There So Many Programming Languages?

» Pretty clear that requirements in various domains are completely different

» Hopefully it’s clear that integrating all of the properties into one language (and having that language do a good job with all requirements) is difficult (and likely impossible)

Given all these programming languages, why the need for new languages?

For a Computer Language to Be Used…

» Someone has to design it: usually only one or very few people, relatively inexpensive

» Someone has to build a compiler: usually 10-20 people for a really large compiler, relatively inexpensive

The Real Cost: All the Users and Educating Them

- » Thousands, hundreds of thousands, potentially millions of developers: time and money spent teaching them language is very large cost
- » Cost of classes, textbooks, etc
- » AND since many programmers teach themselves, their time cost is a real economic cost
- » SO, think about cost of educating a million programmers, it's a significant economic investment.

Consequences

- » Even making small changes to an established language used by millions is very expensive
  - » Even small change to interface to compiler is costly in terms of educating users
  - » So, rate of change of language slows.
  - » For most popular languages, this causes ossification

Another Consequence of Cost of Learning Language

- Paradoxically, it’s relatively ease to start a new language
  - Zero users, so zero training costs at the beginning
  - Even with few users, costs of teaching them changes in language are not very high
  - So new language can adapt much more quickly to changing situations (i.e., just not very expensive to experiment with a new language)

So…

- When a programmer is choosing between an established language that doesn't change quickly, and a new language that has the potential for real gains in productivity in the short term (and at not too much time and expense), then she's *going to go* with the new language.
- When? When there is some void that needs to be filled (i.e., current programming languages not meeting some requirement)

Voids?

- We are in the middle of an information revolution, with new application domains arising all the time.
- New kinds of programming emerge every few years.
  - Right now: Mobile applications (and attendant technology to support mobile computing)
  - A few years ago: the Internet (the phase during which Java came into being)

Summary: Programming Language Economics

- Languages are adopted to fill a void
  - Enable a previously difficult/impossible application
  - Orthogonal to language design quality (almost)

- Observation: Programmer training is the dominant cost of software development
  - Languages with many users are replaced rarely
  - Popular languages become ossified
  - But easy to start in a new niche . . .

One More Consequence

- New programming languages tend to look a lot like old programming languages
  - Leverage the knowledge programmers already have about the old language
  - Classic Example: Java vs. C++
    - Made entry to Java easy for C++ programmers

Topic: Language Design

- No universally accepted metrics for design
  - Not among programmers
  - Not among researchers in programming language design

Topic: Language Design

- Claim: “A good language is one people use”
  - Not. By this criteria, Visual Basic is the world’s best programming language
  - Besides: lot’s goes into language adoption (as we’ve seen) besides technical excellence
  - Language addresses niche -> becomes established in niche -> then inertia sets in
    - Thus Fortran and Cobol are still used, and languages that we could, if starting over today, design much better.

Language Evaluation Criteria

Characteristic	Criteria		
	Readability	Writeability	Reliability
Simplicity	*	*	*
Data types	*	*	*
Syntax design	*	*	*
Abstraction		*	*
Expressivity		*	*
Type checking			*
Exception handling			*

History of Ideas: Abstraction

- Abstraction = detached from concrete details
- Abstraction necessary to build software systems
- Modes of abstraction
  - Via languages/compilers:
    - Higher-level code, few machine dependencies
  - Via subroutines
    - Abstract interface to behavior
  - Via modules
    - Export interfaces; hide implementation
  - Via abstract data types
    - Bundle data with its operations

History of Ideas: Types

- Originally, few types
  - FORTRAN: scalars, arrays
  - LISP: no static type distinctions
- Realization: Types help
  - Allow the programmer to express abstraction
  - Allow the compiler to check against many frequent errors
  - Sometimes to the point that programs are guaranteed "safe"
- More recently
  - Lots of interest in types
  - Experiments with various forms of parameterization
  - Best developed in functional programming

History of Ideas: Reuse

- Reuse = exploit common patterns in software systems
  - Goal: mass-produced software components
  - Reuse is difficult

- Two popular approaches
  - Type parameterization (List(int), List(double))
  - Classes and inheritance: C++ derived classes
  - Combined in C++, Java

- Inheritance allows
  - Specialization of existing abstraction
  - Extension, modification, hiding behavior

Trends

- Language design
  - Many new special-purpose languages
  - Popular languages to stay
- Compilers
  - More needed and more complex
  - Driven by increasing gap between
    - new languages
    - new architectures
  - Venerable and healthy area

Why Study Languages and Compilers ?

5. Increase capacity of expression

4. Improve understanding of program behavior

3. Increase ability to learn new languages

2. Learn to build a large and reliable system

1. See many basic CS concepts at work

Cool Overview

- Classroom Object Oriented Language

- Designed to
  - Be implementable in a short time
  - Give a taste of implementation of modern
    - Abstraction
    - Static typing
    - Reuse (inheritance)
    - Memory management
    - And more …

- But many things are left out

A Simple Example

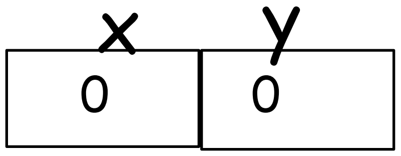
```
class Point {
     x : Int ← 0;
     y : Int ← 0;
};
```

- Cool programs are sets of class definitions
  - A special class **Main** with a special method **main**
  - No separate notion of subroutine
- class = a collection of attributes and methods
- Instances of a class are objects

Cool Objects

```
class Point {
        x : Int ← 0;
        y : Int; (* use default value *)
};
```

- The expression "new Point" creates a new object of class Point
- An object can be thought of as a record with a slot for each attribute



Methods

- A class can also define methods for manipulating the attributes

```
class Point {
     x : Int ← 0;
     y : Int ← 0;
     movePoint(newx : Int, newy : Int): Point {
        { x ← newx;
          y ← newy;
          self;
        } -- close block expression
     }; -- close method
}; -- close class
```

- Methods can refer to the current object using self

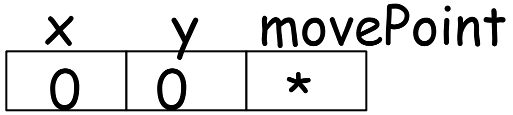
Information Hiding in Cool

- Methods are global
- Attributes are local to a class
  - They can only be accessed by the class's methods
- Example:

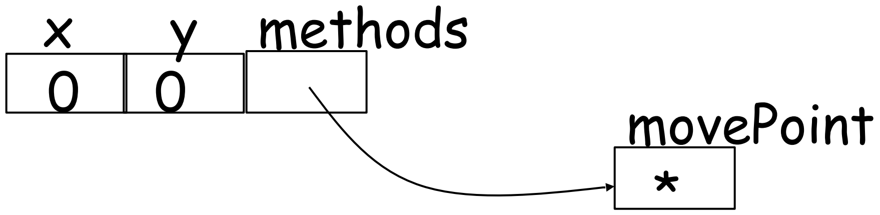
```
class Point {
    . . .
    x () : Int { x };
    setx (newx : Int) : Int { x ← newx };
};
```

Methods

- Each object knows how to access the code of a method
- As if the object contains a slot pointing to the code



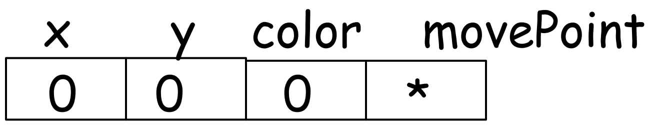
- In reality implementations save space by sharing these pointers among instances of the same class



Inheritance

- We can extend points to colored points using subclassing => class hierarchy

```
class ColorPoint inherits Point {
   color : Int ← 0;
   movePoint(newx : Int, newy : Int): Point {
     { color ← 0;
       x ← newx; y ← newy;
       self;
     }
   };
};
```



Cool Types

- Every class is a type
- Base classes:

Int	for integers
Bool	for boolean values: true, false
String	for strings
Object	root of the class hierarchy

- All variables must be declared
  - compiler infers types for expressions

Cool Type Checking

```
x : A;
x ← new B;
```

- Is well typed if A is an ancestor of B in the class hierarchy
  - Anywhere an A is expected a B can be used
- Type safety:
  - A well-typed program cannot result in runtime type errors

Method Invocation and Inheritance

- Methods are invoked by dispatch

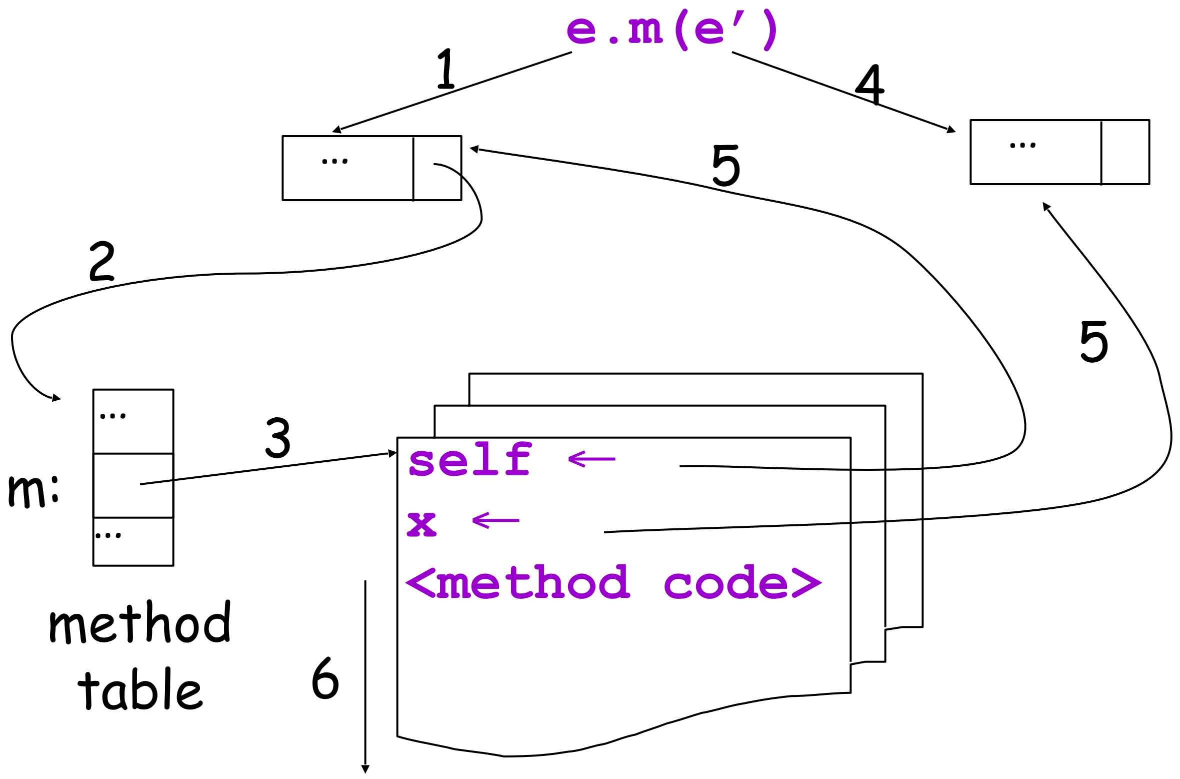
- Understanding dispatch in the presence of inheritance is a subtle aspect of OO languages

```
p : Point;
p ← new ColorPoint;
p.movePoint(1,2);
```

  - p has static type Point
  - p has dynamic type ColorPoint
  - p.movePoint must invoke the ColorPoint version

Method Invocation

- Example: invoke one-argument method m



1. Eval. e
2. Find class of e
3. Find code of m
4. Eval. argum.
5. Bind self and x
6. Run method

Other Expressions

- Expression language
  - every expression has a type and a value

– Loops:	while E loop E pool
– Conditionals	if E then E else E fi
– Case statement	case E of x : Type $\Rightarrow$ E; … esac
– Arithmetic, logical operations	
– Assignment	x $\leftarrow$ E
– Primitive I/O	out_string(s), in_string(), …

- Missing features:
  - arrays, floating point operations, exceptions, …

Cool Memory Management

- Memory is allocated every time new is invoked
- Memory is deallocated automatically when an object is not reachable anymore
  - Done by the garbage collector (GC)
  - There is a Cool GC

Course Project

- A complete compiler
  - Cool ==> MIPS assembly language
  - No optimizations
- Split in 4 programming assignments (PAs)
- There is adequate time to complete assignments
  - But start early and please follow directions
- Individual or team
  - max. 2 students (but adhere to the Shaw Constraint (SC): can only work with a given partner on one project).