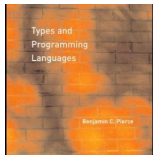
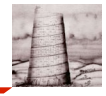
CS 565: Programming Languages

Lecture 1



Administrivia



Who am I?

Course web page

Office hours

Main text

- Types and Programming Languages, B. Pierce, MIT Press

Course Work

Lectures

Homeworks

- *Small & frequent*

Programming exercises?

- *none*

Midterm

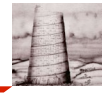
Final

with additional qualifying exam questions

Project

- *Work in groups of ~4*
- *Significant intellectual & programming challenge*

Prerequistes



Programming experience/maturity

- Exposure to various language constructs

  Java, ML, Lisp, Prolog, C
- Undergraduate compilers and/or PL class

  CS 352 and/or CS456 or equivalent

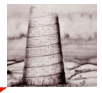
Mathematical maturity

- Familiarity with first-order logic, set theory, graph theory, induction

Most important

- Intellectual curiosity and creativity

Resources



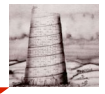
Web page for text

- http://www.cis.upenn.edu/~bcpierce/tapl

Proceedings of conferences

- POPL, PLDI, ICFP, OOSPLA, ECOOP ...

Motivation



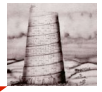
Prove specific facts about programs

- Verify correctness
  - Important in mission-critical systems
- Safety or isolation properties
- Need an unambiguous vocabulary

Understand specific language features

- Better language design
- Guide improvements in implementations

Goals

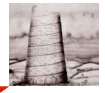


A more sophisticated appreciation of programs, their structure, and the field as a whole

- Viewing programs as rich, formal, mathematical objects, not mere syntax
- Define and prove rigorous claims about a program's meaning and behavior
- Develop sound intuitions to better judge language properties

Develop tools to be better programmers, designers and computer scientists

Topics



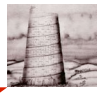
Semantic formalisms, λ-calculus, introduction to types

Simply-typed λ-calculus, records, references, subtyping, object-based programming

Polymorphism, abstract data types, advanced topics (e.g., concurrency, linearity, ...)

Run-time systems, garabage collection, concurrency & multi-threading, synchronization

Language Design

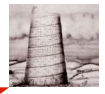


Tower of Babel

- Applications often have distinct (and conflicting) needs
- AI (Lisp, Prolog, Scheme)
- Scientific computing (Fortran)
- Business (Cobol)
- Systems programming (C)
- Scripting (Perl, Javascript)
- Distributed computation (Java)
- Special-purpose (.....)

Important to understand differences and similarities among different language features

Paradigms



Imperative (Fortran, Algol, C, Pascal)

- Designed around a notion of a program store
- Behavior expressed in terms of transformations on the store

Functional (Lisp, ML, Scheme, Haskell)

- Programs described in terms of a collection of functions
- "Pure" functional languages are state-free

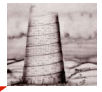
Logic (Prolog)

- Programs described in terms of a collection of logical relations

Concurrent (Fortran90, CSP, Linda)

Special purpose (TeX, Postscript, HTML)

Metrics



No universally accepted criteria

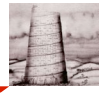
The most popular languages are not necessarily the best ones

- Consider Cobol or JCL (Job Control Language)
- Although, aren't notions of superiority highly subjective?

General characteristics

- Simplicity and "elegance" (orthogonality)
- Readability
- Safety
- Programming-in-the-large
- Efficiency
- Abstraction

Case studies: Lisp 1.5



Based on λ-calculus

Key aspect of the calculus is notion of substitution of free variables:

```
function f(args)= ....x....
```

Suppose `x` is not included in args. Where should the binding for `x` be constructed?

- At the point where f is defined (lexical scoping)
- At the points where f is applied (dynamic scoping)

Lisp chose dynamic scoping, even though it is widely agreed today that lexical scoping is the more sensible choice. When do these distinctions arise? Why are the differences important?

Case Studies: Java Array Types

One of Java's design mistakes is the subtyping rule for arrays

- Given Object types `T` and `T'`, the array subtyping rule is

```
T <: T'  ⇒  T[] <: T'[]
```

if `T` is a subtype of `T'` then `T[]` is a subtype of `T'[]`

- Example: `String <: Object` thus
  `String[] <: Object[]`

Case Studies: Java Array Types

Static typing guarantees that if we have declaration

```
T v;
```

then the following holds at all times

```
v instanceof T
```

This is good because it guarantees that type errors do not occur at run-time.

Arrays break static type safety

Case Studies: Java Array Types

The following is a static error:

```
Thread[] appThreads = new Thread[10];
appThreads[0] = "badaboom";                    //type error
```

But the following is perfectly fine:

```
Object[] appThreads = new Thread[10];
appThreads[0] = "badaboom";
runFirstThread(  (Thread[]) appThreads ) ;

void runFirstThread( Thread[] o ) {
 o[0].run();                                   //runtime type error
```


Case Studies: Java Array Types

```
Object[] appThreads = new Thread[10];  //OK Thread[] <: Object[]
appThreads[0] = "badaboom";            //OK String <: Object
runFirstThread((Thread[]) appThreads); //appThreads instanceof Thread[]

void runFirstThread( Thread[] o ) {
   o[0].run();                         //OK o[0] is declared to be a Thread
```

The problem is the subtyping rule

Array stores are checked at run-time by the JVM.

Run-time overhead can be high; `instanceof` is linear in the number of super types.

Case Studies: Java Array Types

Why have that rule ?

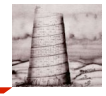
To write `sort(Object[])` once rather than have to rewrite it for each new class!

- but subtype polymorphism is a poor substitute for parametric polymorphism

The right solution is genericity…

```
sort(<A implements Comparable<A>>)
```

Lessons



Language design is as much about safety as it is about efficiency and expressiveness.

Need tools and frameworks to reason about and compare different language features and designs:

- untyped λ-calculus as a universal computation language. Precisely define its behavior using appropriate semantic models
- typed λ-calculi to express safety and abstraction properties