Programming Languages

Pragmatics first
Questions

What the Course Will Cover

Examples: The January 2001 Exam

Top-Down Overview
Pragmatics

Instructor

Brook Conner, conner@cs.nyu.edu, me@nellardo.com
Office hours 5-6 Tuesdays or by appt. Room 401.

Grader/TA

Sean MacLaughlin, maclaughlin@cs.nyu.edu

Email list

http://www.cs.nyu.edu/mailman/listinfo/g22_2110_001_su02

Texts

Policies
Texts

Required


Also covers many compiler-related topics

Recommended

*Types and Programming Languages*, Pierce.


Policies

Sign up for the email list

Weekly assignments
  Will be on website by start of class
  Due at start of class following week
  Prose answers on paper
  Executable code in email

Discussion of topics is cool; Discussion of answers is not
Questions?
What the Course Will Cover

Abstraction

Survey of the semantics of specific PLs

Identification and analysis of categories of semantics

Language mechanisms and design patterns
Abstraction

Design of PL is design of abstraction mechanism

Machine -> assembly; Assembly -> compilation, structured
Structured -> Functional, Logic, OO

Abstraction: Remove detail to simplify reasoning

Modeling: Creating an abstraction of a system to solve a problem

No PL is a “Silver Bullet”

If a PL abstracts out a crucial aspect of your system, it is a poor modelling tool for that system

Cannot forget the context: The problem to solve
Survey of the semantics of specific PLs

Not lexicography

- Regular expressions, choice of keywords, number formats
- Detail — province of compilers course

Not syntax

- Braces or keywords? “Goto considered harmful”, interpreted or compiled
- Food for flame-wars and parser generators

“Abstracting the abstractions”

- Eliding detail of syntax to study the semantics, the mechanics
Identification and analysis of categories of semantics

Not Data-Driven vs Control-Driven

Dead issue: Real algorithms, real systems are both
May apply to a single mechanism, but not language as whole
Some languages have little or no distinction

Mutability
Aggregation
Uniformity

But first, two examples: Self and Haskell
Self: The Power of Simplicity

Objects are a set of names; Each name points to another object; Pair is a slot

Send a message to an object

Looks for matching name and evaluates result

Some objects eval by returning themselves
Others send more messages

Primitive assignment changes object pointed to

Some slots are parents: if object can’t find slot matching message, searches recursively in parents
Self: Implications

Object-oriented without classes, predefined control structures, data types, etc.

Anything can change at any time

All primitive ops (int add, etc.) are external libraries

Design patterns re-create higher-level semantics

  If-then-else is implemented by two objects, true and false

  *true ifThen: foo. will evaluate foo*

  *false ifThen: foo. will not*

  Assign to slots *true and/or false* and change everything
Haskell: Purely Functional, Purely Declarative

Everything is a value

No value is ever changed; only create new ones

Functions are values, too (and only have 1 arg)

No value is evaluated until it absolutely must be

Strictly typed; but types are inferred by compiler

Pattern-matching to choose among alternate functions
Haskell: Implications

Infinite data structures

All fibonacci numbers:

\[ \text{fibs} = 0 : 1 : \text{zipWith (+)} \text{ fibs} (\text{tail fibs}) \]

Functions > 1 arg are fns that return fns

\[ \text{zipWith func (a:at) (b:bt) = func a b : \text{zipWith func at bt}} \]
\[ \text{zipWith _ _ _} = [] \]
\[ \text{addLists} = \text{zipWith '+} \]

Sometimes, div by 0 is safe....

\[ \text{if True then 0 else 1/0} \]
Mutability

_Name:_ syntactic identifier plus a referent

Id may be lexical (e.g., variable name)
Id may be more complex (e.g., if-then-else)

To what level can _names_ change their _referent_?

_Extreme:_ Self

Almost anything can change at any time

_Extreme:_ Haskell

Never change, only hide
Aggregation

What support is there for combining names?
Is there grouping? Is there sharing?
How flexible is the sharing?
Extremes: Assembly
Less extreme: C
Some sharing: CLU, Modula
OO: C++, Java, Obj-C, Lisp/CLOS, Smalltalk
Pervasive grouping and sharing: Self, Haskell
Uniformity

Are name referents platform-specific?

Are all names treated the same?

Extreme: Assembly

Lots of built-in names: C, C++, Perl

Some first-class names: Java, Scheme, Lisp

Pervasive paradigm with syntactic sugar: Haskell

Single paradigm: Self
Language mechanisms and design patterns

*Design Patterns*: Gamma, Johnson, Helm, Vlissides

Bits of design that work to solve sub-problems

What is mechanism in one language is pattern in another

Mechanism: C++ class

Pattern: C struct with array of function pointers

   Exactly how early C++ compilers worked
Why Use Patterns

Start from very simple language, very simple semantics

Compare mechanisms of other languages by building patterns in simpler language

Enable meaningful comparisons between language mechanisms
Example: Jan 01 Exam

Question 2. Given C++ class....

    Constructor: name=new char[strlen(foo)];

a. Write its destructor

    Given answer: if [name!=0] then delete name;
    Correct answer: delete name[];
Example: Jan 01 Exam

Question 3. Given ML func meld

Haskell version: zipWith

b. Rewrite in imperative lang of your choice

Given answer in C++

```cpp
template <class t1, class t2, class t3> class meld {
  typedef &t1 (*FP)(&t2, &t3);
  public &list<t1> meld(FP p, &list<t2> l1, &list<t3> l3); }
```

Problem one: it doesn’t parse — refs in wrong place

Problem two: not a function

Problem three: doesn’t work if it uses references
  references fall out of scope and become invalid; if
  new’ed, this is a monster memory leak
Example: Jan 01 Exam

Question 3: Given meld

c. What happens if the lists are different lengths? What’s the best way to address this?

Given answer: Check lists for equal lengths and throw exception if not equal

Alternate: meld for the length of the shorter and return that zipWith does this as is
Next Week

Assignment: See web

Reading:

PL Pragmatics, Ch 1. Ch 2 if you’re unfamiliar with lexing and parsing

PL Pragmatics, Ch 10.