CS 242

Programming Languages

John Mitchell

Course web site: http://www.stanford.edu/class/cs242/

All about me ...

John C. Mitchell

Professor of Computer Science

Research Interests: Computer security: access control, cryptographic protocols and mobile code security. Programming languages, type systems, object systems, and formal methods. Applications of logic to CS.

B.S. Stanford University; M.S., Ph.D. MIT.

- How I spend my time
  - Working with graduate students
  - Writing papers, going to conferences, giving talks
  - Departmental committees (hiring, curriculum, ...)
  - Teaching classes
  - Conferences, journals, consulting, companies, ...

Goals

- Programming Language Culture
  - A language is a "conceptual universe" (Perlis)
    - Learn what is important about various languages
    - Understand the ideas and programming methods
  - Understand the languages you use (C, C++, Java) by comparison with other languages
  - Appreciate history, diversity of ideas in programming
  - Be prepared for new problem-solving paradigms
- Critical thought
  - Properties of language, not documentation
- Language *and* implementation
  - Every convenience has its cost
    - Recognize the cost of presenting an abstract view of machine
    - Understand trade-offs in programming language design

Transference of Lang. Concepts

- Parable
  - I started programming in 1970's
    - Dominant language was Fortran; no recursive functions
  - My algorithms and data structure instructor said:
    - Recursion is a good idea even though inefficient
    - You can use idea in Fortran by storing stack in array
  - Today: recursive functions everywhere
- Moral
  - World changes; useful to understand many ideas
- More current example: function passing
  - Pass functions in C by building your own closures, as in STL "function objects"

Alternate Course Organizations

- Language-based organization
  - Algol 60, Algol 68, Pascal
  - Modula, Clu, Ada
  - Additional languages grouped by paradigm
    - Lisp/Scheme/ML for functional languages
    - Prolog and Logic Programming
    - C++, Smalltalk and OOP
    - Concurrency via Ada rendez-vous

My opinion:

Algol/Pascal/Modula superseded by ML
Lisp/Scheme ideas also in ML
OOP deserves greater emphasis

~~For comparison, see Sethi's book ...~~

Alternate Course II

- Concept-based organization
  - Use single language like Lisp/Scheme
  - Present PL concepts by showing how to define them
- Advantages:
  - uniform syntax, easy to compare features
- Disadvantages
  - Miss a lot of the culture associated with languages
  - Some features hard to add
    - Type systems
    - Program-structuring mechanisms
    - Works best for "local" features, not global structure

Examples: ~~Abelson/Sussman, Friedman et al.~~

Organization of this course

- Programming in the small
  - Cover traditional Algol, Pascal constructs in ML
    - Block structure, activation records
    - Types and type systems, ...
  - Lisp/Scheme concepts in ML too
    - higher-order functions and closures, tail recursion
    - exceptions, continuations
- Programming in the large
  - Modularity and program structure
  - Specific emphasis on OOP
    - Smalltalk vs C++ vs Java
    - Language design and implementation

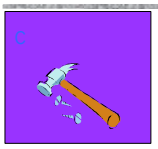
Course Organization (cont'd)

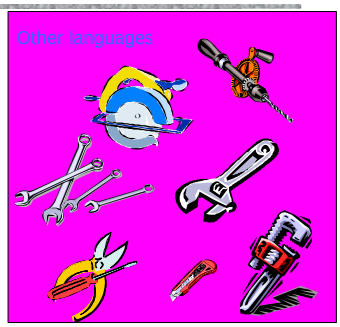
- Concurrent and distributed programming
  - General issues in concurrent programming
  - Actor languages: an attempt at idealization
  - Java threads

But what about C?

- Important, practical language
- But, most of you think C all the time
- We discuss other languages, you compare them to C in your head as we go (and in homework)

Programming language toolsets





If all you have is a hammer,
then everything looks like a nail.

Digression

- Current view from carpentry

"A hammer is more than just a hammer. It's a personal tool that you get used to and you form a loyalty with. It becomes an extension of yourself."

http://www.hammernet.com/romance.htm

First half of course

- Lisp (2 lectures)
- Foundations (2 lectures)
  - Lambda Calculus
  - Denotational Semantics
  - Functional vs Imperative Programming
- Conventional prog. language concepts (6 lectures)
  - ML/Algol language summary (1 lecture)
  - Types and type inference (1 lecture)
  - Block structure and memory management (2 lectures)
  - Control constructs (2 lectures)

---------- Midterm Exam ----------

Second half of course

- Modularity and data abstraction (1 lecture)
- Object-oriented languages (6 lectures)
  - Introduction to objects (1 lecture)
  - Simula and Smalltalk (2 lectures)
  - C++ (1.5 lectures)
  - Java (1.5 lectures)
- Concurrent and distributed programming (1 lecture)
- Conclusions and review (1 lecture)

---------- Final Exam ----------

General suggestions

- Read ahead
  - Some details are only in HW and reading
- There is something difficult about this course
  - May be hard to understand homework questions
    - Thought questions: cannot run and debug
    - May sound like there is no right answer, but some answers *are* better than others
  - Many of you may be used to overlooking language problems, so it takes a few weeks to see the issues

TAU BETA PI ONLINE COURSE GUIDE

Mitchell, Autumn 1998-99

- This is a fun course and its not too hard. Some of the homework questions take a lot of thought. Beware.
- Fundamentals and theory bog down the class a bit, but overall an interesting class and Mitchell an interesting and funny teacher.
- Not too tough, can be tricky. ... the material is interesting and makes you think about things in a different way...
- Take it it's very useful. Now it's much easier for me to figure out what is going in my program or trace the errors of a program and choose the right language for certain tasks.

Course Logistics

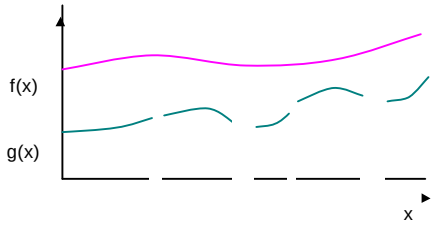
- Homework and Exams
  - HW on Tuesdays
  - Midterm and Final: dates are set
  - Honor Code, Collaboration Policy
- TA's, Office hours, Email policy, ...
- Section
  - Thursday 7-8 PM
  - Optional discussion and review; no new material
    - Not broadcast but notes may be available electronically (?)
- Reading material
  - Book on order. Let's hope it arrives.

Look at web site...

Foundations: Partial,Total Functions

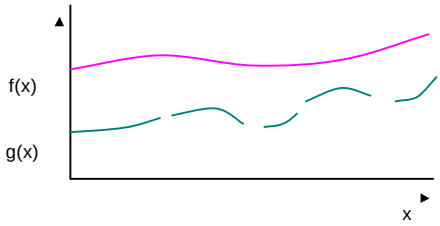
- Value of an expression may be undefined
  - Undefined operation, e.g., division by zero
    - 3/0 has no value
    - implementation may halt with error condition
  - Nontermination
    - f(x) = if x=0 then 1 else f(x-2)
    - this is a *partial* function: not defined on all arguments
    - cannot be detected at compile-time; this is halting problem
  - These two cases are
    - "Mathematically" equivalent
    - Operationally different

Partial and Total Functions



- Total function: f(x) has a value for every x
- Partial function: g(x) does not have a value for every x

Functions and Graphs



- Graph of f = { ⟨x,y⟩ | y = f(x) }
- Graph of g = { ⟨x,y⟩ | y = g(x) }

Mathematics: a function is a set of ordered pairs (graph of function)

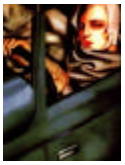
Partial and Total Functions

- Total function $f:A \to B$ is a subset $f \subseteq A \times B$ with
  - For every $x \in A$, there is some $y \in B$ with $\langle x,y \rangle \in f$ (total)
  - If $\langle x,y \rangle \in f$ and $\langle x,z \rangle \in f$ then $y=z$ (single-valued)
- Partial function $f:A \to B$ is a subset $f \subseteq A \times B$ with
  - If $\langle x,y \rangle \in f$ and $\langle x,z \rangle \in f$ then $y=z$ (single-valued)
- Programs define partial functions for two reasons
  - partial operations (like division)
  - nontermination

    f(x) = if x=0 then 1 else f(x-2)

Halting Problem

Entore Buggati: "I build cars to go, not to stop."



Self-Portrait in the Green Buggati (1925)
Tamara DeLempicka

Computability

- Definition

  A function f is computable if there is a program P that computes f, i.e., for any input x, the computation P(x) halts with output f(x)

- Terminology

  Partial recursive functions
  = partial functions (integers to integers) that are computable

Halting function

- Decide whether program halts on input
  - Given program P and input x to P,

$$Halt(P,x) = \begin{cases} \text{yes} & \text{if } P(x) \text{ halts} \\ \text{no} & \text{otherwise} \end{cases}$$

Clarifications

Assume program P requires one string input x
Write P(x) for output of P when run in input x
Program P is string input to *Halt*

Fact: There is no program for *Halt*

Unsolvability of the halting problem

- Suppose P solves variant of halting problem
  - On input Q, assume

$$P(Q) = \begin{cases} \text{yes} & \text{if } Q(Q) \text{ halts} \\ \text{no} & \text{otherwise} \end{cases}$$

- Build program D
  - D(Q) = run forever if Q(Q) halts; halt if Q(Q) runs forever
- Does this make sense? What can D(D) do?
  - If D(D) halts, then D(D) runs forever.
  - If D(D) runs forever, then D(D) halts.
  - *CONTRADICTION:* program P must not exist.

Main points about computability

- Some functions are computable, some are not
  - Halting problem
- Programming language implementation
  - *Can* report error if program result is undefined due to division by zero, other undefined basic operation
  - *Cannot* report error if program will not terminate

Announcements

- Homework grader?
  - Send email to cs242 email list
- Something for fun
  - Nominate theme song for programming language or course topic
- Questions???