CSE 413: Programming Languages and their Implementation

Hal Perkins
Spring 2011
Today’s Outline

• Administrative Info
• Overview of the Course
• Introduction to Scheme
Registration

• Please sign up on info sheet being passed around if you’re still trying to get in
• We’ll see what we can do, but no promises (also depends on how many requests there are)
Who, Where & When

• **Instructor**
  » Hal Perkins (perkins@cs.washington.edu)

• **Teaching Assistants**
  » Nathan Armstrong, Liem Dinh, Jiayun (Gloria) Guo, Chanel Huang
  
  Office hours & locations tba, etc.

• **Lectures**
  » MWF 2:30-3:20, BAG 261
Web Page

• All info is on the CSE 413 web:

   http://www.cs.washington.edu/education/courses/413/11sp

• Look there for schedules, contact information, assignments, links to discussion boards and mailing lists, etc.
CSE 413 E-mail List

• If you are registered for the course you will be automatically added.

• E-mail list is used for posting important announcements by instructor and TAs

• You are responsible for anything sent here
  » Mail to this list is sent to your UW email address
CSE 413 Discussion Board

• Use the Catalyst GoPost message board to stay in touch outside of class
  » Staff will watch and contribute too

• Use:
  » General discussion of class contents
  » Hints and ideas about assignments (but not detailed code or solutions)
  » Other topics related to the course
Course Computing

• College of Arts & Sciences Instructional Computing Lab (aka Math Science Computing Labs)
• Or work from home – all software is freely available
  » See links on the course web
Grading: Estimated Breakdown

• Approximate Grading:
  » Homework + Project: 55%
  » Midterm: 15% (TBA, est. 5/6 in class)
  » Final: 25% (Tue. June 7, 2:30-4:20)
  » Participation 5%

• Assignments:
  » Weights may differ to account for relative difficulty of assignments
  » Assignments will be a mix of shorter written exercises and longer programming projects
Deadlines & Late Policy

• Assignments generally due Thursday evenings via the web
  » Exact times and dates given for each assignment

• Late policy: 4 late days per person
  » At most 2 on any single assignment
  » Used only in integer units
  » For group projects, both students must have late days available and both are charged if used
  » Don’t burn them up early!!
Academic (Mis-)Conduct

• You are expected to do your own work
  » Exceptions (group work), if any, will be clearly announced

• Things that are academic mis-conduct:
  » Sharing solutions, doing work for or accepting work from others
  » Searching for solutions on the web
  » Consulting solutions to assignments or projects from previous offerings of this course

• Integrity is a fundamental principle in the academic world (and elsewhere) – we and your classmates trust you; don’t abuse that trust
1) **Information Sheet (aka Assignment #0):**
   Bring to lecture on Friday April 1
2) **Download and Install DrRacket**
   » (and play with it!)
3) **Reading:** See “Scheme Resources” on Web page
4) **Assignment #1:** (coming soon!)
Reading

• No required text – we’ll make some suggestions as we go along
• Other references available from course web page

• Check “Functional Programming & Scheme” Link for:
  » Notes on Scheme
  » Revised\(^5\) Report on the Algorithmic Language Scheme (R5RS)
    • The language definition: this is your friend!
  » Link to Structure and Interpretation of Computer Programs (Abelson, Sussman, & Sussman)
    • Detailed textbook from MIT – overkill for us, but fantastic!
Tentative Course Schedule

- Week 1: Scheme
- Week 2: Scheme
- Week 3: Scheme
- Week 4: Scheme wrapup/intro to Ruby
- Weeks 5-6: Object-oriented programming and Ruby; scripting languages
- Weeks 7-9: Language implementation, compilers and interpreters
- Week 10: garbage collection; special topics
Now where were we?

- Programming Languages
- Their Implementation
Why Scheme?

- Focus on “functional programming” because of simplicity, power
- Stretch our brains – different ways of thinking about programming and computation
  - Often a good way to think if stuck in C/Java/…
- Let go of Java/C/… for now
  - Easier to approach functional programming on its own terms
  - We’ll make the connections back to what you’ve seen before later in the quarter
Functional Programming

- Programming consists of defining and evaluating functions
- No side effects (assignment)
  » An expression will always yield the same value when evaluated (referential transparency)
- No loops (use recursion instead)

- Scheme includes assignment and loops but they are not needed except in specific circumstances and we will avoid them
Primitive Expressions

• constants
  » integer :
  » rational :
  » real :
  » boolean :

• variable names (symbols)
  » Names can contain almost any character except white space and parentheses
  » Stick with simple names like value, x, iter, ...
Compound Expressions

• Either a combination or a special form

1. Combination: (operator operand operand …)
   » there are quite a few pre-defined operators
   » We can define our own operators

2. Special form
   » keywords in the language
   » eg, define, if, cond
Combinations

• (operator operand operand …)

• this is *prefix* notation, the operator comes first

• a combination always denotes a procedure application

• the operator is a symbol or an expression, the applied procedure is the associated value
  » +, -, abs, my-function
  » characters like * and + are not special; if they do not stand alone then they are part of some name
Evaluating Combinations

• To evaluate a combination
  » Evaluate the subexpressions of the combination
  » Apply the procedure that is the value of the leftmost subexpression (the operator) to the arguments that are the values of the other subexpressions (the operands)

• Examples (demo)
Evaluating Special Forms

• Special forms have unique evaluation rules

• `(define x 3)` is an example of a special form; it is not a combination
  » the evaluation rule for a simple define is "associate the given name with the given value"

• There are a few more special forms, but there are surprisingly few of them compared to other languages
Procedures
Recall the *define* special form

- Special forms have unique evaluation rules
- `(define x 3)` is an example of a special form; it is not a combination
  » the evaluation rule for a simple define is "associate the given name with the given value"
Define and name a variable

- \( \text{define (name (expr))} \)
  - define - special form
  - name - name that the value of \( expr \) is bound to
  - expr - expression that is evaluated to give the value for name

- define is valid only at the top level of a <program> and at the beginning of a <body>
Define and name a procedure

- `(define (⟨name⟩ ⟨formal params⟩) ⟨body⟩)`
  - `define` - special form
  - `name` - the name that the procedure is bound to
  - `formal params` - names used within the body of procedure
  - `body` - expression (or sequence of expressions) that will be evaluated when the procedure is called.

  » The result of the last expression in the body will be returned as the result of the procedure call
Example definitions

(define pi 3.1415926535)

(define (area-of-disk r)
  (* pi (* r r)))

(define (area-of-ring outer inner)
  (- (area-of-disk outer)
      (area-of-disk inner)))
Defined procedures are "first class"

- Compound procedures that we define are used exactly the same way the primitive procedures provided in Scheme are used
  - names of built-in procedures are not treated specially; they are simply names that have been pre-defined
  - you can't tell whether a name stands for a primitive (built-in) procedure or a compound (defined) procedure by looking at the name or how it is used
Booleans

- Recall that one type of data object is boolean
  - `#t` (true) or `#f` (false)
- We can use these explicitly or by calculating them in expressions that yield boolean values
- An expression that yields a true or false value is called a predicate
  - `< 5 5` =>
  - `(> pi 0)` =>
Conditional expressions

• As in all languages, we need to be able to make decisions based on inputs and do something depending on the result
Special form: **cond**

- \((\text{cond } \langle claus{e_1}\rangle \langle claus{e_2}\rangle \ldots \langle claus{e_n}\rangle)\)
- each clause is of the form
  - » \((\langle \text{predicate}\rangle \langle \text{expression}\rangle)\)
- the last clause can be of the form
  - » \((\text{else } \langle \text{expression}\rangle)\)
Example: sign.scm

; return the sign of x as -1, 0, or 1

(define (sign x)
  (cond
    ((< x 0) -1)
    ((= x 0) 0)
    ((> x 0) +1)))
Special form: \texttt{if}

- \((\texttt{if } \langle \texttt{predicate} \rangle \langle \texttt{consequent} \rangle \langle \texttt{alternate} \rangle)\)

- \((\texttt{if } \langle \texttt{predicate} \rangle \langle \texttt{consequent} \rangle)\)
Examples : abs.scm

; absolute value function
(define (abs a)
Logical composition

- \((\text{and } \langle e_1 \rangle \langle e_2 \rangle \cdots \langle e_n \rangle)\)
- \((\text{or } \langle e_1 \rangle \langle e_2 \rangle \cdots \langle e_n \rangle)\)
- \((\text{not } \langle e \rangle)\)

- Scheme interprets the expressions \(e_i\) one at a time in left-to-right order until it determines the correct value
in-range.scm

; true if val is lo <= val <= hi

(define (in-range lo val hi)
  (and (<= lo val)
       (<= val hi)))