CSE 413: Programming Languages and their Implementation

Hal Perkins
Winter 2019
Today’s Outline

• Administrative info
• Overview of the course
• Introduction to Racket
Registration

Not registered yet?

• Please watch for changes in registration and grab an empty slot when one shows up

• We won’t attempt to manage wait lists or add codes for now
Who, Where & When

• Instructor: Hal Perkins (perkins@cs.washington.edu)
• TAs: Jack Eggleston, Aaron Johnston, Johnny Wu, Nate Yazdani
• Office hours: will set up and announce shortly
• Lectures: MWF 2:30-3:20, CMU 120
• No sections, but would people be interested in some sort of (semi-)formal work sessions?
  – What if we attach 1 credit hour to it?
Course Web

• All info is on the CSE 413 web:

  www.cs.uw.edu/413

• Look there for schedules, contact information, lecture materials, assignments, links to discussion boards and mailing lists, etc.
CSE 413 Discussion Board

• We’re using a Google group
  – Log in with your “UW Google Credentials”
  – Link on the course home page

• Join in, help each other out, stay connected outside of class
CSE 413 E-mail List

• If you are registered for the course you are automatically subscribed
• Used for posting important announcements by instructor and TAs
• You are responsible for anything sent here
  – Mail to this list is sent to your designated UW email address
Course Computing

• All software is freely available and can be installed anywhere you want
  – Links on the course web

• Also should be available in the College of Arts & Sciences Instructional Computing Lab
  – Let us know if there are problems
Grading: Estimated Breakdown

• Approximate Grading:
  – Homework: 55%
  – Midterm: 15% (in class, prob. Fri. Feb. 15)
  – Final: 25% (Tue. March 19?, 2:30 pm)
  – Other ≤5% (citizenship, effort, ...)

• Assignments:
  – Weights will differ depending on difficulty
  – Assignments will be a mix of shorter written exercises and shorter/longer programming projects
Deadlines & Late Policy

• Assignments submitted online, graded, and feedback returned via GradeScope
  – Due @11pm
  – Most due Tuesday evenings, a few other nights
  – Calendar has likely schedule; might change some

• Late policy: 4 “late days” for entire quarter
  – At most 2 on any single assignment
  – Used only in integer, 24-hour units
  – Don’t burn them up early!!
Academic (Mis-)Conduct

• You are expected to do your own work
  – Exceptions, if any, will be clearly announced

• Things that are academic misconduct:
  – Sharing solutions, doing work for others, accepting work from others including have someone “walk you through” the details
  – Copying solutions found on the web
  – Consulting solutions from previous offerings of this course
  – etc. Will not attempt to provide exact legislation and invite attempts to weasel around the rules

• Integrity is a fundamental principle in the academic world (and elsewhere) – we and your classmates trust you; don’t abuse that trust

• You must know the course policy– Read It! (on the web)
Working With Colleagues

• “Do your own work” does not mean “lock yourself in a windowless room”. Learning from each other and from the course staff is a good thing; sharing ideas and talking is a good thing; finding useful resources is a good thing.

– Representing something that you didn’t do as your own is not.
  • OK?
Gadgets (1)

• Gadgets reduce focus and learning
  – Bursts of info (e.g. emails, IMs, etc.) are *addictive*
  – Heavy multitaskers have more trouble focusing and shutting out irrelevant information
  – Seriously, you will learn more if you use *paper* instead!!!
Gadgets (2)

• So how should we deal with laptops/phones/etc.?
  – Just say no!
  – No open gadgets during class (really!)*
    • *Exceptions possible in cases where it actually makes sense – discuss with instructor
  – Urge to search? – ask a question! Everyone benefits!!
  – You may close/turn off your electronic devices now
  – Pull out a piece of paper and pen/pencil instead 😊

• We will post code samples and transcripts of demos; but you’ll want to have your own notes about key points and ideas
  – Class should not be the same as watching videos with brains clicked off 😊
Reading

• No required $$$ textbook
• Good resources on the web
• “Functional Programming/Racket” link on course web:
  – Course notes! (also linked to calendar – *read them!*)
  – Racket documentation
  – How to Design Programs
    • Intro textbook using Scheme
  – Structure and Interpretation of Computer Programs
    • Fantastic, classic intro CS book from MIT. Some good examples here that are directly useful
Tentative Course Schedule

• Week 1: Functional Programming/Racket
• Week 2: Functional Programming/Racket
• Week 3: Functional Programming/Racket
• Week 4: FP wrapup, environments, lazy eval
• Weeks 5-6: Object-oriented programming and Ruby; scripting languages
• Weeks 7-9: Language implementation, compilers and interpreters
• Week 10: garbage collection; special topics
Work to do!

• Download Racket and install

• Run DrRacket and verify facts like 1+1=2
  – Which, in racket is `(eqv? (+ 1 1) 2) 😊

• Learn your way around the course web and linked resources
  – Especially: *read* the Racket lecture notes that go with the first lectures
Now where were we?

• Programming Languages

• Language Implementation
Why Functional Programming?

• Focus on “functional programming” because of simplicity, power, elegance
• Stretch our brains – different ways of thinking about programming and computation
  – Often a good way to think even if stuck with C/Java/…
• Now mainstream – lambdas/closures in Javascript, C#, Java 8, C++11; functional programming is the “secret sauce” in Google’s infrastructure; ...
• Let go of Java/C/… for now
  – Easier to approach functional prog. on its own terms
  – We’ll make connections to other languages as we go
Scheme / Racket

• Scheme: The classic functional language
  – Enormously influential in education, research

• Racket
  – Modern Scheme dialect with some changes/extras
  – DrRacket programming environment (was DrScheme for many years)

• Expect your instructor to say “Scheme” accidentally at times
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)

• Racket/Scheme/Lisp include assignment and loops but they are not needed and we won’t use
  – i.e., you will “lose points”, as the saying goes 😊
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 sumsq, x, iter, same?, ...
Compound Expressions

• Either a combination or a special form

1. Combination: (operator op1 op2 ...)
   – there are a lot of pre-defined operators
   – We can define our own operators

2. Special form
   – “keywords” in the language
   – eg, define, if, cond
   – have non-standard evaluation rules (more later)
Combinations

• (operator operand1 operand2 ...)  

• 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, new-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
    • All of them, including the operator – it’s an expression too!
  – 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" or, more concisely, "bind the value to the name"
  - All special forms do something different from simple evaluation of a value from (evaluated) operands
- There are a few more special forms, but there are surprisingly few 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”, i.e., “bind the value to the name”
Bind a value to a variable

• (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>
  – We will only use it at top-level
Bind a procedure value (!) to a name

- `(define (⟨name⟩ ⟨params⟩) ⟨body⟩)`
  - define - special form
  - name - the name that the procedure is bound to
  - formal parameters - names used within the body of procedure, bound when procedure is called
  - 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”

- Procedures that we define are used exactly the same way as the primitive procedures provided in Racket
  - names of built-in procedures are not special; they are simply names that have been pre-defined
  - you can't tell whether a name stands for a primitive (built-in) procedure or one we’ve defined by looking at the name or how it is used
  - [Disclaimer: This is almost but not always strictly true in Racket]
Booleans

• 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
  
  #t  =>
  (< 5 5)  =>
  (> pi 0)  =>
Conditional expressions

• As in all languages, we need to be able to make decisions based on values

• In Racket it’s not “if this is true, do that else do something else”

• Instead, we have conditional expressions. The value of a conditional expression is the value of one of its subexpressions – which one depends on the value(s) of other expression(s)
Special form: if

\[(\text{if } \langle e1 \rangle \langle e2 \rangle \langle e3 \rangle)\]

Evaluation:
- Evaluate \(\langle e1 \rangle\)
- If true, evaluate \(\langle e2 \rangle\) to get the if value
- If false, evaluate \(\langle e3 \rangle\) to get the if value

Example: \((\text{if } (\langle \langle < x y \rangle \rangle x y)\)
**Special form: cond**

\[(\text{cond } \langle\text{clause1}\rangle \langle\text{clause2}\rangle \ldots \langle\text{clausen}\rangle)\]

- each clause has the form
  \[\langle\text{predicate}\rangle \langle\text{expression}\rangle\]
  - (Racket allows us to use \[\] and ( ) interchangeably, which can make things more readable)

- the last clause can be
  \[\langle\text{else}\rangle \langle\text{expression}\rangle\]
Example: sign.scm

; return the sign of x: -1, 0, 1
(define (sign x)
    (cond
        [(< x 0) -1]
        [(= x 0) 0]
        [(> x 0) +1])))
Logical composition

(and \langle e_1 \rangle \langle e_2 \rangle \ldots \langle e_n \rangle)
(or \langle e_1 \rangle \langle e_2 \rangle \ldots \langle e_n \rangle)
(not \langle e \rangle)

• Racket evaluates the expressions ei 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)))
To Be Continued...

- For more information about Racket/Scheme, refer to notes on the Racket pages of the course web & reference material linked there

- More demos/examples in the next several lectures, very little PowerPoint, if any