CSE 413: Programming Languages and their Implementation

Hal Perkins

Autumn 2008

Today’s Outline

- Administrative Info
- Overview of the Course
- Introduction to Scheme

Staff

- Instructor
  - » Hal Perkins (perkins@cs.washington.edu)
- Teaching Assistant
  - » Laura Marshall (lmarsh16@cs.washington.edu)

Web Page

- All info is on the web page for CSE 413
  (or at least will be once things are a bit further along…)
  - » http://www.cs.washington.edu/413
  - » also known as

    http://www.cs.washington.edu/education/courses/413/08au
- 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 registered.
- 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 uwnetid

CSE 413 Discussion Board

- The course has a Catalyst GoPost message board
- Use it 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)
  - » Basement of Communications building: B-022/027
  - » http://depts.washington.edu/aslab
- Or work from home – all software available free
  - » See links on the course web

Grading: Estimated Breakdown:

- Approximate Grading:
  - » Homework + Project: 55%
  - » Midterm: 15% (TBA, in class)
  - » Final: 25% (Thursday December 11 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 will be 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

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

Homework for Today!!

1) **Assignment #1:** (posted in the next day or so)
2) **Information Sheet (aka Assignment #0)**: Bring to lecture on Friday Sept 26
3) **Download and Install Dr. Scheme** (and play with it!)
4) **Reading:** See "Scheme Resources" on Web page

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:
  - » More notes on Scheme
  - » *Revised^5 Report on the Algorithmic Language Scheme (R5RS)*
    - Section 2
  - » Link to *Structure and Interpretation of Computer Programs* (Abelson, Sussman, & Sussman)
    - Sections 1-1.1.5

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

What is this course about?

- Programming Languages
- Their Implementation

Why Study Programming Languages?

- Become a better software engineer
  - » Understand how to use language features
  - » Appreciate implementation issues
- Better background for language selection
  - » Familiar with range of languages
  - » Understand issues/advantages/disadvantages
- Better able to learn languages:
  - » You will learn many over your career

Why Study Compilers/Interpreters?

- Better understanding of implementation issues in programming languages:
  - » How is “this” implemented?
  - » Why does “this” run so slowly?
- Translation appears many places:
  - » Processing command line parameters
  - » Converting files/programs from one language/format to another

Why are there so many (1,000s) Programming Languages?

- **Evolution**: random coding -> structured programming -> OO programming
- **Special Purposes**: Lisp for symbols, Snobol for strings, C for systems, Prolog for relationships
- **Personal Preference**: Programmers have their own personal tastes

What Makes a Programming Language Successful?

- Expressive power (more suited to a particular task)
- Easy to use (teaching/learning)
- Ease of implementation (easy to write a compiler/interpreter for)
- Good compilers (Fortran)
- Economics, patronage (Cobol, Ada)

- Donald Knuth:

» *Programming is the art of telling another human being what one wants the computer to do.*

Programming Domains

- Scientific applications:
  - » Using the computer as A large calculator
  - » FORTRAN, mathematica
- Business applications:
  - » Data processing and business procedures
  - » COBOL, some PL/I, spreadsheets
- Systems programming:
  - » Building operating systems and utilities
  - » C, c++

Programming Domains (2)

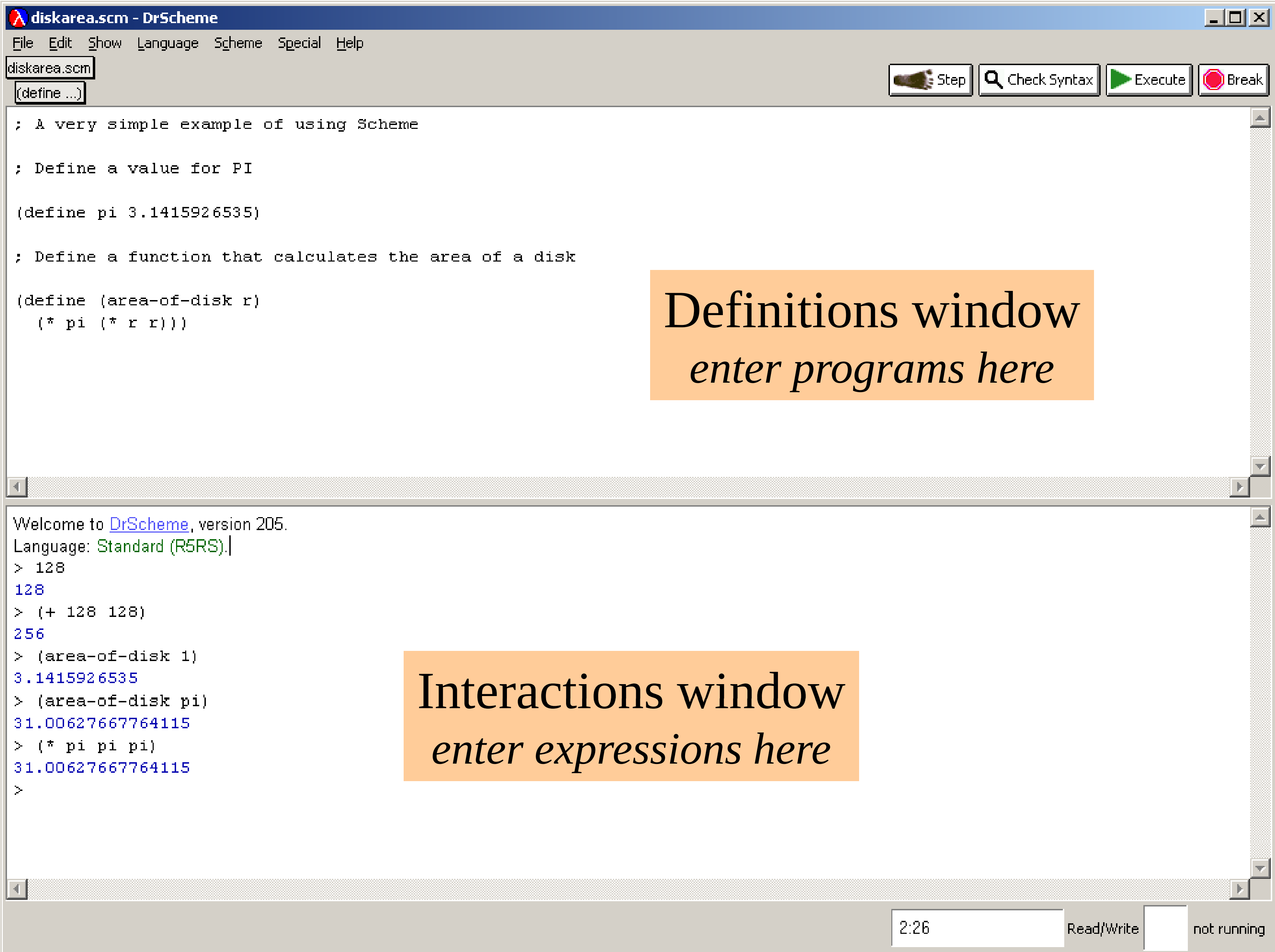
- Parallel programming:
  - » Parallel and distributed systems
  - » Ada, CSP, Erlang, functional map/reduce (Google)
- Artificial intelligence:
  - » Uses symbolic rather than numeric computations
  - » Lists as main data structure, flexibility (code = data)
  - » Lisp 1959, prolog 1970s
- Scripting languages:
  - » A list of commands to be executed
  - » UNIX shell programming, awk, tcl, perl

Programming Domains (3)

- Education:
  - » Languages designed to facilitate teaching
  - » Pascal, BASIC, logo
- Special purpose:
  - » Other than the above...
  - » Simulation
  - » Specialized equipment control
  - » String processing
  - » Visual languages

Why Scheme?

- The simplicity of the language lets us work on problem solving, rather than just syntax issues
- Structure of the language lets us see that the structure of C/Java/Basic is not the only way to express problem solutions
- Stretch our brains
  - » study more than one language paradigm and study the relationship between design paradigms
  - » Recursive programming is an essential part of a programmer's toolkit


diskarea.scm - DrScheme
File Edit Show Language Scheme Special Help
diskarea.scm
(define ...)
Step
Check Syntax
Execute
Break
; A very simple example of using Scheme
; Define a value for PI
(define pi 3.14159265335)
; Define a function that calculates the area of a disk
(define (area-of-disk r)
(* pi (* r r)))
Definitions window
enter programs here
Welcome to DrScheme, version 205.
Language: Standard (R5RS).
> 128
128
> (+ 128 128)
256
> (area-of-disk 1)
3.14159265335
> (area-of-disk pi)
31.00627667764115
> (* pi pi pi)
31.00627667764115
>
Interactions window
enter expressions here
2:26
Read/Write
not running

Definitions window

- Define programs in the definitions window
  - » Save the contents of the window to a file using menu item file - save definitions as …
  - » Load existing files with menu item file - open
  - » Execute the contents of the definitions window by clicking on the “run" button
  - » Check and highlight syntax by clicking on the "check syntax" button
  - » Re-indent all with control-i

Interactions Window

- Evaluate simple expressions directly in the Interactions window
- Position the cursor after the ">", then type in your expression
  - » DrScheme responds by evaluating the expression and printing the result
  - » recall previous expression with escape-p
- Expressions can reference symbols defined when you executed the Definitions window

Think functionally

- Procedural programming
  - » The order of assignments changes the operation of the program because the state is changed by assignment
- Functional programming (Scheme)
  - » Computation is a sequence of function definitions and evaluations
  - » Core is free of side-effects (assignment)
  - » Referential transparency: An expression will always yield the same value when evaluated
    - Not true in presence of side-effects

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 subexpresions (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 some more special forms which we will encounter, but there are surprisingly few of them compared to other languages

Procedures

References

- Section 15.5, *Concepts of Programming Languages*
- Section 4.1, *Revised^5 Report on the Algorithmic Language Scheme (R5RS)*

- For more help:
  - » Sections 1.1.6-1.1.8, *Structure and Interpretation of Computer Programs* (Abelson, Sussman, & Sussman)

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

- **`(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
  - » **`#t`** =>
  - » **`(< 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

Predicate **Consequent**

Special form: **cond**

- **(cond** $\langle clause_1 \rangle \langle clause_2 \rangle$ **...** $\langle clause_n \rangle$**)**
- each clause is of the form
  - » **(**$\langle predicate \rangle \langle expression \rangle$**)**
- the last clause can be of the form
  - » **(else** $\langle expression \rangle$**)**

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: **`if`**

- **`(if`** ⟨*predicate*⟩ ⟨*consequent*⟩ ⟨*alternate*⟩**`)`**
- **`(if`** ⟨*predicate*⟩ ⟨*consequent*⟩ **`)`**

Examples : abs.scm

```
; absolute value function
(define (abs a)
```

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`** $\langle e \rangle$**`)`**

- Scheme interprets the expressions e_i one at a time in left-to-right order until it can tell the correct answer

in-range.scm

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

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