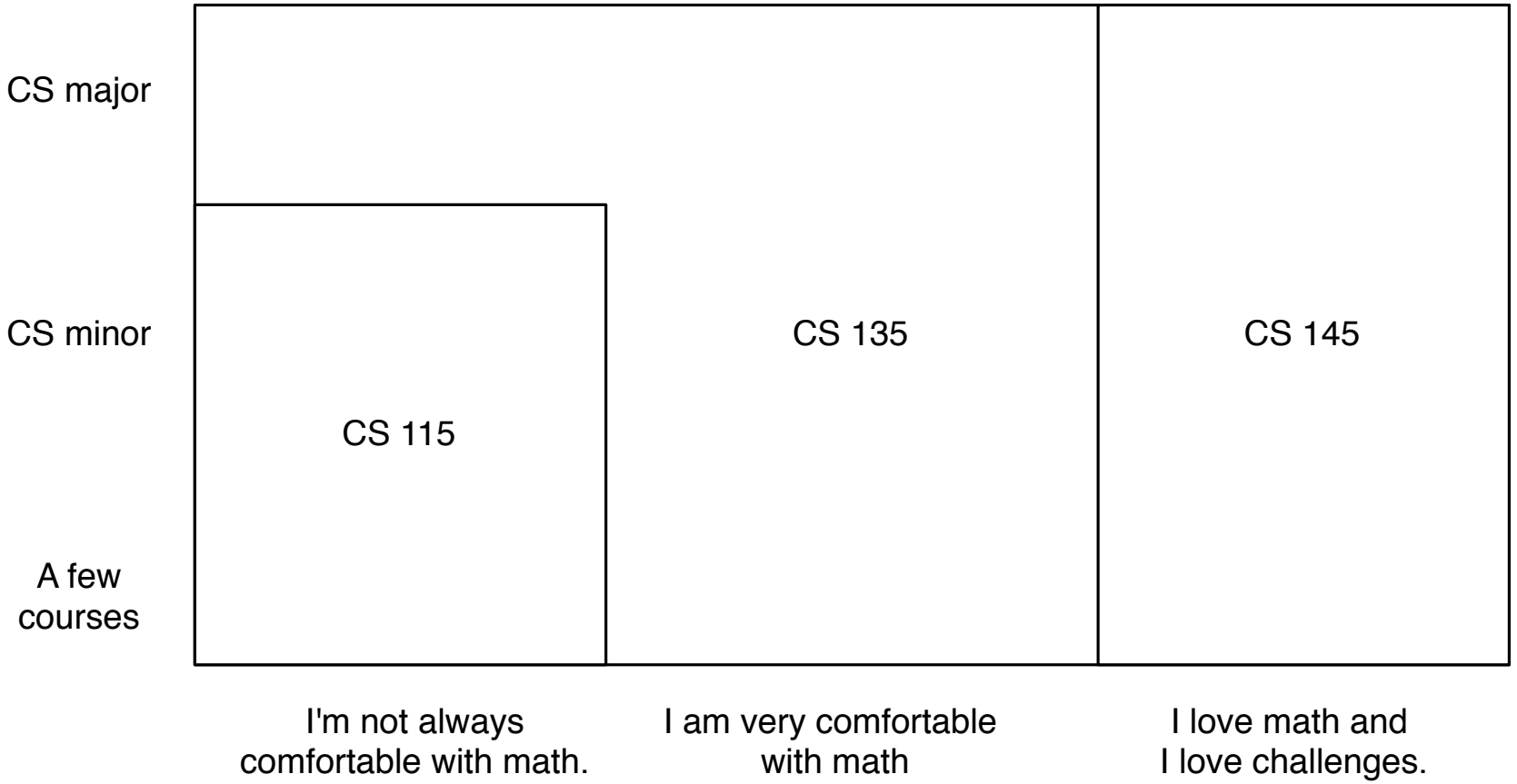
Welcome to CS 115 (Fall 2012)

For all course staff (including instructors, tutors, instructional assistants, and the instructional support coordinator) see

`http://www.student.cs.uwaterloo.ca/~cs115/personnel`

Am I in the right course?



Factors: interest in CS, comfort with math

Course components

Lectures: Two 80 minutes lectures per week

Textbook: “How to Design Programs” (HtDP) by Felleisen, Flatt, Findler, Krishnamurthi (`http://www.htdp.org`)

Presentation handouts: available at Campus Copy in the Math building and online

Web page (main information source):
`http://www.student.cs.uwaterloo.ca/~cs115`

Labs: 90 minutes using DrRacket v4.1 or higher (`http://www.DrRacket.org`)

Assignments: Roughly weekly, worth 30%

Exams: 25% midterm, 45% final; you must pass the weighted exam component to pass the course.

Work for your first week:

- Read the Web pages (see menu bar).
- Read the Survival Guide (bound in with the presentation handouts and on the ”Resources” page).

Suggestions for success

Take notes in lecture.

Keep up with the readings (keep ahead if possible).

Attend weekly labs.

Start assignments early.

Integrate exam study into your weekly routine.

Keep on top of your workload.

Avoiding pitfalls

Work efficiently – don't waste time.

Follow our advice on approaches to writing programs (e.g. design recipe, templates).

Visit office hours to get help.

Go beyond the minimum required (e.g. extra exercises).

Maintain a "big picture" perspective: look beyond the immediate task or topic.

Programming practice

Labs prepare you for assignments.

Assignments prepare you for exams.

You must do your own work in this course; read the section on plagiarism in the CS 115 Survival Guide.

Role of programming languages

When you click an icon for an application, a program is run in machine language.

Machine language is designed for particular hardware, rather than being human-friendly and general.

People use high-level languages to express computation naturally and generally.

Programming language design

Imperative:

- frequent changes to data
- examples: machine language, Java, C++

Functional:

- computation of new values, not transformation of old ones
- examples: Excel formulas, LISP, ML, Haskell, Mathematica, XSLT, R (used in STAT 231)
- more closely connected to mathematics
- easier to design and reason about programs

Why start with Scheme?

- used in education and research for over twenty years
- minimal but powerful syntax
- tools easy to construct from a few simple primitives
- DrRacket, a helpful environment for learning to program
- nice transition to an imperative language, Python, in CS 116

The DrRacket environment

- designed for education, powerful enough for “real” use
- sequence of language levels keyed to textbook
- includes good development tools
- two windows: Interactions and Definitions
- Interactions window: a read-evaluate-print loop (REPL)

Course goals

CS 115 **isn't** a course in programming in Scheme.

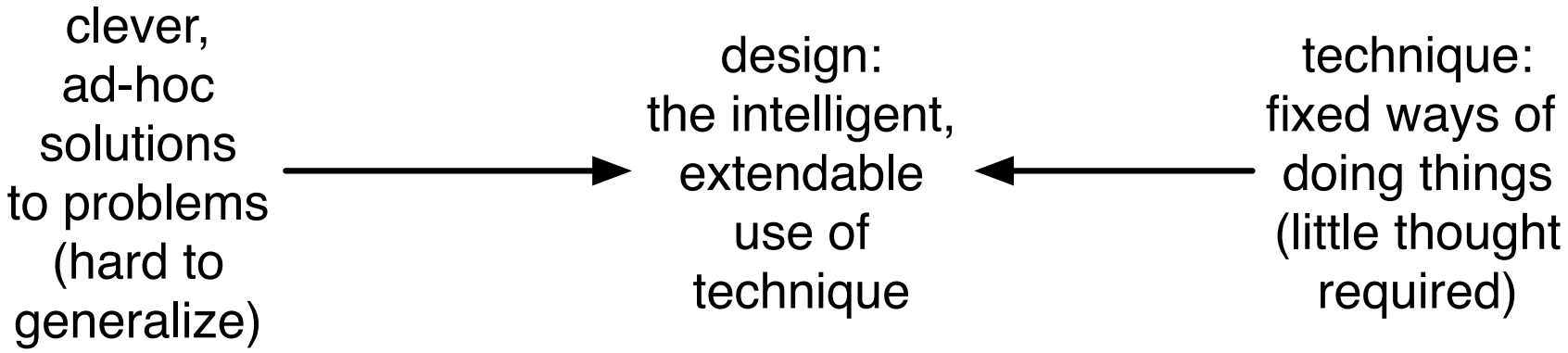
CS 115 **is** an introduction to computational thought (thinking precisely and abstractly).

CS 115 uses only a very small subset of Scheme (plus some additional features not found in standard Scheme).

CS 116 will introduce more features of Scheme, and make the transition to an industry-oriented imperative language, Python.

Knowing two different programming paradigms will make it easier for you to extract the higher-level concepts being taught.

We will cover the whole process of designing programs.



Careful use of design processes can save time and reduce frustration.

Themes of the course

- design (the art of creation)
- abstraction (finding commonality, not worrying about details)
- refinement (revisiting and improving initial ideas)
- syntax (how to say it), expressiveness (how easy it is to say and understand), and semantics (the meaning of what's being said)
- communication (in general)

Simple Scheme programs

Using Scheme is like using a super pocket calculator.

Each expression in your program is evaluated.

The simplest example is a number.

- Integers in Scheme are unbounded.
- Rational numbers are represented exactly.
- Expressions whose values are not rational numbers are flagged as being **inexact**. We will not use these much (if at all).

We can write more interesting programs by using functions.

Functions in mathematics

A function generalizes similar expressions.

$3^2 + 4 \cdot 3 + 2$

$6^2 + 4 \cdot 6 + 2$

$7^2 + 4 \cdot 7 + 2$

These are generalized by the function
$f(x) = x^2 + 4x + 2$.

A **function definition** consists of

- the name of the function,
- its **parameters**, and
- a mathematical expression using the parameters.

Examples:

$f(x) = x^2$

$g(x, y) = x - y$

A **function application** supplies **arguments** for the parameters.

Examples:

$g(3, 1)$

$g(f(2), g(3, 1))$

The mathematical expression is **evaluated** by substitution.

The arguments are substituted into the expression in place of the parameters.

Example:

$g(3, 1) = 3 - 1 = 2$

We say that g **consumes** 3 and 1 and **produces** 2.

Function applications in Scheme

In a Scheme function application, parentheses are put around the function name and the arguments.

To translate from mathematics to Scheme

- move '(' to before the name of the function, and
- replace each comma with a space.

$g(3, 1)$ becomes (g 3 1)

$g(f(2), g(3, 1))$ becomes (g (f 2) (g 3 1))

There is one set of parentheses for each function application.

Mathematical expressions

For some mathematical functions, function applications are written in a different way.

In $3 - 2 + 4/5$, the symbols for addition, subtraction, and division are put between numbers.

Precedence rules (division before addition, left to right) specify the order of operation.

To associate a function with arguments in a way different from that given by the rules, parentheses are required: $(6 - 4)/(5 + 7)$.

Converting an expression into Scheme

As before, we put parentheses around the function and arguments.

$3 - 2$ becomes $(-\ 3\ 2)$

$3 - 2 + 4/5$ becomes $(+\ (-\ 3\ 2)\ (/\ 4\ 5))$

$(6 - 4)(3 + 2)$ becomes $(*\ (-\ 6\ 4)\ (+\ 3\ 2))$

In Scheme, parentheses are used to associate arguments with functions, and are no longer needed for precedence.

Extra parentheses are harmless in mathematical expressions, but they are harmful in Scheme. **Only use parentheses when necessary.**

Built-in functions

Scheme has many built-in functions, such as familiar functions from mathematics, other functions that consume numbers, and also function that consume other types of data.

You can find the quotient of two integers using quotient and the remainder using remainder.

```
(quotient 75 7)
(remainder 75 7)
```

Scheme expressions causing errors

What is wrong with each of the following?

- `(* (5) 3)`
- `(+ (* 2 4)`
- `(5 * 14)`
- `(* + 3 5 2)`
- `(/ 25 0)`

Preventing errors

Syntax: the way we're allowed to express ideas

Syntax error when an expression cannot be interpreted using the syntax rules: `(+ - 3)`

Semantics: the meaning of what we say

Semantic error when an expression cannot be reduced to a value by application of substitution rules: (sqrt 5 9)

'Trombones fly hungrily' has correct syntax (spelling, grammar) but no meaning.

English syntax rules (e.g. a sentence has a subject, a verb, and an object) are not rigidly enforced.

Scheme syntax rules are interpreted by a machine. They are rigidly enforced.

If you break a syntax rule, you will see an error message.

Syntax rule: a **function application** consists of the function name followed by the expressions to which the function applies, all in parentheses, or (functionname exp1 . . . expk).

We need rules for function names and expressions to complete this.

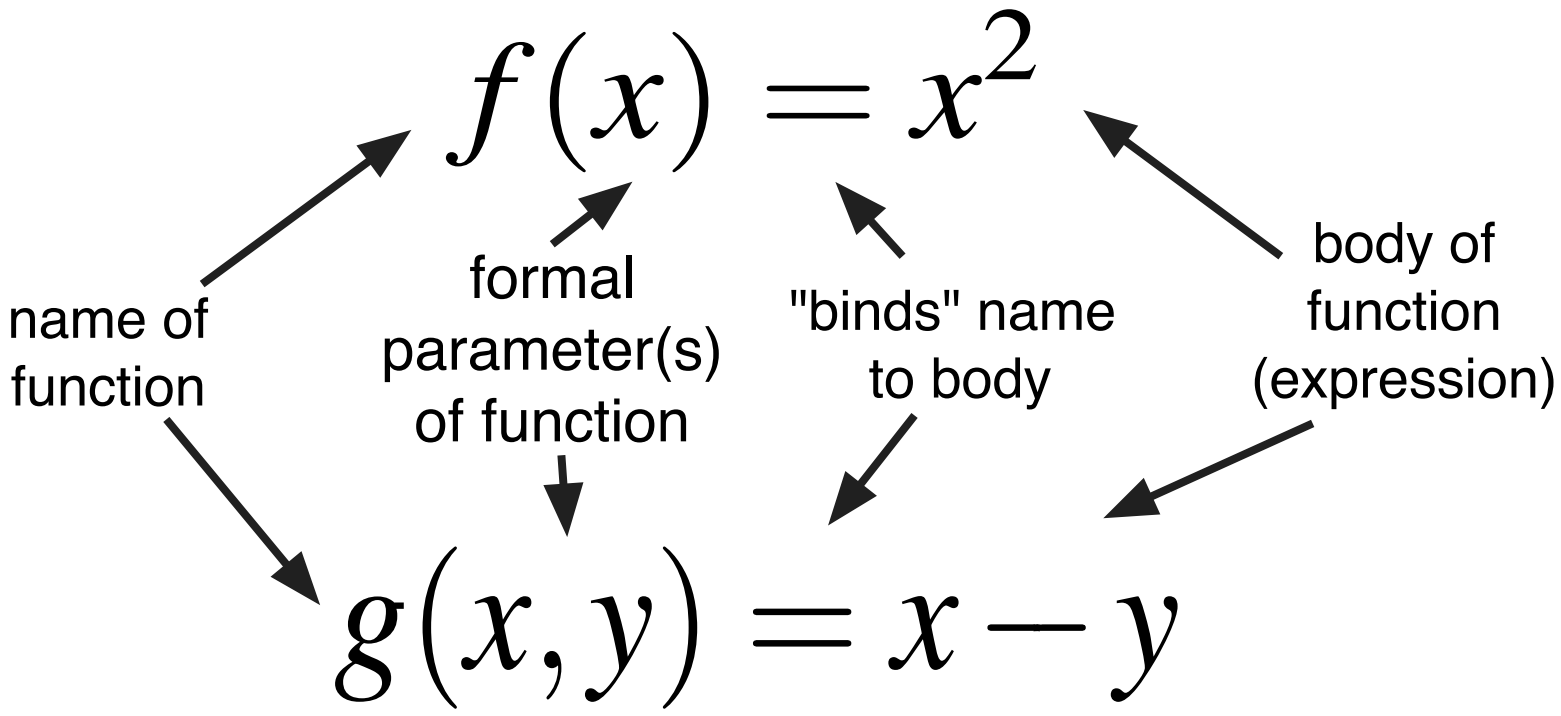
So far Scheme is no better than a calculator or a spreadsheet.

A spreadsheet formula applies functions to data.

A spreadsheet application provides many built-in functions, but typically it is hard to construct your own.

Scheme provides a modest number of built-in functions, but makes it easy to construct your own.

Defining functions in mathematics



Important observations:

- Changing names of parameters does not change what the function does: $f(x) = x^2$ and $f(z) = z^2$ have the same behaviour.
- The same parameter name can be used in different functions, as in f and g.
- The order of arguments must match the order of the parameters in the definition of the function: $g(3, 1) = 3 - 1$ but $g(1, 3) = 1 - 3$.

Defining functions in Scheme

Our definitions $f(x) = x^2$ and $g(x, y) = x - y$ become

```
(define (f x) (* x x))
(define (g x y) (- x y))
```

define is a **special form**; it looks like a Scheme function, but not all of its arguments are evaluated.

It **binds** a name to an expression (which uses the parameters that follow the name).

A function definition consists of

- a name for the function,
- a list of parameters, and
- a single “body” expression.

The body expression typically uses the parameters together with other built-in and user-defined functions.

To give names to the function and parameters, we use identifiers.

Syntax rule: an **identifier** starts with a letter, and can include letters, numbers, hyphens, underscores, and a few other punctuation marks. It cannot contain spaces or any of ( ) , { } [ ] ‘ ’ “ ”.

Syntax rule: **function definition** is of the form (define (id1 . . . idk) exp), where exp is an expression and each id is an identifier.

We can fix the syntax rule for a function application to say that a function name is a built-in name or an identifier.

Important observations:

- Changing names of parameters does not change what the function does: (define (f x) (* x x)) and (define (f z) (* z z)) have the same behaviour.
- The same parameter name can be used in different functions, as in f and g.
- The order of arguments must match the order of the parameters in the definition of the function: (g 3 1) produces 2 but (g 1 3) produces -2.

DrRacket's Definitions window

- can accumulate definitions and expressions
- Run button loads contents into Interactions window
- can save and restore Definitions window
- provides a Stepper to let one evaluate expressions step-by-step (to be shown later)
- features: error highlighting, subexpression highlighting, syntax checking

Constants in Scheme

The definitions $k = 3$ and $p = k^2$ become

```
(define k 3)
(define p (* k k))
```

The first definition binds the identifier k to the value 3.

In the second definition, the expression (* k k) is first evaluated to give 9, and then p is bound to that value.

Syntax rule: a **constant definition** is of the form (define id exp).

Defining constants allows us to give meaningful names to values (e.g. interest-rate, passing-grade, and starting-salary).

Usefulness of constants:

- Making programs easier to understand.
- Making future changes easier.
- Reducing typing.

The text uses the term variable to refer to constants (but they can't change until CS 116).

Programs in Scheme

A **Scheme program** is a sequence of definitions and expressions.

- The definitions are of functions and constants.
- The expressions typically involve both user-defined and built-in functions and constants.

Programs may also make use of **special forms** (which look like functions, but don't necessarily evaluate all their arguments).

Identifiers and binding

Which of the following uses of x are allowed?

Names of parameters of two different functions:

```
(define (f x) (* x x))
(define (g x y) (- x y))
```

A constant and a parameter of a function:

```
(define x 3)
(define (f x) (* x x))
```

Constant and in body of a function:

```
(define x 3)
(define (g y) (* x y))
```

Two constants:

```
(define x 4)
(define x 5)
```

A constant and a name of a function:

```
(define x 4)
(define (x y) (* 5 y))
```

A name and a parameter of different functions:

```
(define (x y) (* 5 y))
(define (z x) (* 3 x))
```

A name and parameter of the same function:

```
(define (x x) (+ 1 x))
```

The importance of semantics

The English sentence “Cans fry cloud” is syntactically correct but has no semantic interpretation.

The English sentences “Students hate annoying professors”, “I saw her duck”, “You will be very fortunate to get this person to work for you”, and “Kids make nutritious snacks” are all ambiguous; each has more than one semantic interpretation.

We must make sure that our Scheme programs are unambiguous (have exactly one interpretation).

A semantic model

A semantic model of a programming language provides a method of predicting the result of running any program.

A Scheme program is a sequence of definitions and expressions.

Our model involves the simplification of the program using a series of steps (**substitution rules**).

Each step yields a valid Scheme program.

A fully-simplified program is a sequence of definitions and values.

Values

A value is something which cannot be further simplified.

So far, the only things which are values are numbers. As we move through CS115, we will see many other types of values.

For example, 3 is a value, but (+ 3 5) and (f 3 2) are not.

Using substitution

We used substitution to evaluate a mathematical expression:

$$g(f(2), g(3,1)) = g(4, g(3,1)) = g(4,2) = 2$$

There are many possible ways to substitute, such as

$g(f(2), g(3,1)) = f(2) - g(3,1)$ and

$g(f(2), g(3,1)) = g(f(2), 2)$.

We will also use substitution for the Scheme program:

```
(define (f x) (* x x))
(define (g x y) (- x y))
(g (f 2) (g 3 1))
```

Evaluating a Scheme expression

Goal: a single sequence of substitution steps for any expression.

Constraints:

1. We first evaluate the arguments, and then apply the function to the resulting values.

2. When we have the choice among two or more substitutions, we take the **leftmost** one.

These decisions do not change the final result.

We use the 'yields' symbol $\Rightarrow$ to show the result of a single step.

Substitution rules

Built-in function application: use mathematics rules.

$(+\ 3\ 5) \Rightarrow 8$

(quotient 75 7) $\Rightarrow$ 10

Constant: replace the identifier by the value to which it is bound.

(define x 3)

$(*\ \text{x}\ (+\ \text{x}\ 5))$

$\Rightarrow (*\ 3\ (+\ \text{x}\ 5))$

$\Rightarrow (*\ 3\ (+\ 3\ 5))$

$\Rightarrow (*\ 3\ 8) \Rightarrow 24$

Value: no substitution needed.

User-defined function application: for a function defined by (define (f x1 x2 . . . xn) exp), simplify a function application (f v1 v2 . . . vn) by replacing all occurrences of the parameter xi by the value vi in the expression exp.

Tricky points to remember:

- Each vi must be a **value**.
- Replace **all** occurrences in one step.

General rule: Anything that has been fully evaluated (e.g. a function definition) does not need to be repeated at the next step.

```
(define (f x) (* x x))
(define (g x y) (- x y))
(g (f 2) (g 3 1))
⇒ (g (* 2 2) (g 3 1))
⇒ (g 4 (g 3 1))
⇒ (g 4 (- 3 1))
⇒ (g 4 2)
⇒ (- 4 2) ⇒ 2
```

Tracing a program

Tracing: a step-by-step simplification by applying substitution rules.

Any expression that has been fully simplified does not need to appear in the next step of the trace.

Use tracing to check your work for semantic correctness.

If no rules can be applied but the program hasn't been simplified, there is an error in the program, such as (sqr 2 3).

Be prepared to demonstrate tracing on exams.

The Stepper may not use the same rules. Write your own trace to be sure you understand your code.

```
(define (term x y) (* x (sqr y)))
(term (- 5 3) (+ 1 2))
⇒ (term 2 (+ 1 2))
⇒ (term 2 3)
⇒ (* 2 (sqr 3))
⇒ (* 2 9)
⇒ 18
```

The data type image

We will use **data types** other than numbers.

DrRacket also supports images using the `world.ss` teachpack.

The teachpack provides functions that consume and produce images.

```
(define image1 (rectangle 35 50 'solid 'red))
(define image2 (circle 10 'solid 'blue))
(overlay image1 image2)
```

There is more information on the Resources page of the course Web site.

Goals of this module

You should be comfortable with these terms: function, parameter, application, argument, variable, expression, consume, produce, syntax, semantics, special form, bind, identifier, Scheme program.

You should be able to define and use simple arithmetic functions, and to define constants.

You should know how to form expressions properly, and what DrRacket might do when given an expression causing an error.

You should understand the purposes and uses of the Definitions and Interactions windows in DrRacket.

You should be able to argue that your Scheme code is syntactically correct, referring to Scheme syntax rules.

You should understand the basic syntax of Scheme, including rules for function application, function definition, and constant definition.

You should be able to trace the substitutions of a Scheme program, using substitution rules for function applications and uses of constants.