CS115 - Module 1

Cameron Morland

Fall 2017

Welcome to CS 115

Information about the course is on the website:
`https://www.student.cs.uwaterloo.ca/~cs115/`

Contact info for everyone is there, but if in doubt, send me an email:
`cjmorland@uwaterloo.ca`

If you have a laptop with you, please install DrRacket right away: `racket-lang.org`
Your labs will help you get acquainted with DrRacket.

Am I in the right course?

There are alternatives to this course!

CS135 and CS145 cover essentially the same content, but in greater depth.

If you wish to be a CS major, you do not *need* to take 135; you may be able to take CS116 then CS136. But taking 135 may be preferable.

If you are not a math student, you may prefer CS105.

First Week Tasks

Be sure you do all the following:

1. Install DrRacket on your laptop.
2. Find out about your labs, and participate in the first one.
3. From the website `https://www.student.cs.uwaterloo.ca/~cs115/` download the course notes and review the course details, including survival guide, marking scheme, and grade appeals policy.
4. Bookmark the course textbook, `http://www.htdp.org` , and read the appropriate sections.
5. Register your iClicker, and see how clickers affect your grade. Details are on the course website.
6. Complete Assignment 00.

Participation

Your participation mark comes from *clicker* questions in class. The purpose of these questions is encourage participation and provide feedback on your learning.

You receive 2 marks for each correct answer, 1 mark for each incorrect answer. The best 75% of all questions is used to calculate your grade.

You must attend your own section to receive these grades.

Never use another student's clicker.

Register your clicker on the CS115 website, *not at the central UW website!*

Most of your learning comes from struggling with material. You learn little from merely copying work from another.

All assignments are to be done individually.
Don't look at someone else's programs written for an assignment, or show your programs to someone else. Don't search on the web or in books other than the textbook for answers to assignment questions, or even for hints.

Start your assignments early, and bring questions to office hours as soon as possible.
Use labs to get practice.

Computers run only **machine code**. This is machine code:

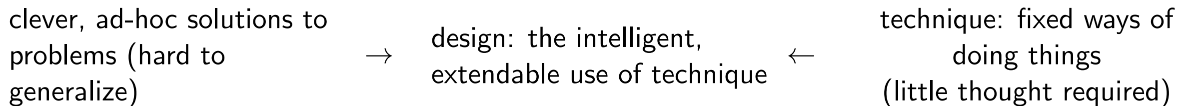
```
6a0:    55                  push   %rbp
6a1:    48 89 e5            mov    %rsp,%rbp
6a4:    48 83 ec 10         sub    $0x10,%rsp
6a8:    89 7d fc            mov    %edi,-0x4(%rbp)
6ab:    89 75 f8            mov    %esi,-0x8(%rbp)
6ae:    89 55 f4            mov    %edx,-0xc(%rbp)
6b1:    83 7d f4 00         cmpl   $0x0,-0xc(%rbp)
6b5:    75 05               jne    6bc <fibonacci+0x1c>
6b7:    8b 45 f8            mov    -0x8(%rbp),%eax
6ba:    eb 1a               jmp    6d6 <fibonacci+0x36>
6bc:    8b 45 f4            mov    -0xc(%rbp),%eax
6bf:    8d 50 ff            lea    -0x1(%rax),%edx
6c2:    8b 4d fc            mov    -0x4(%rbp),%ecx
6c5:    8b 45 f8            mov    -0x8(%rbp),%eax
6c8:    01 c1               add    %eax,%ecx
6ca:    8b 45 f8            mov    -0x8(%rbp),%eax
```

It is easier to write in a higher level language, like C, which is **compiled** to machine code:

```
// fibonacci(previous, latest, count): returns the
//   count-th number following latest in the Fibonacci
//   sequence, where previous is the one before latest.
// fibonacci: Nat Nat Nat -> Nat
// Examples:
// fibonacci(0, 1, 0) => 1
// fibonacci(0, 1, 4) => 5
// fibonacci(5, 8, 1) => 13
int fibonacci(int previous, int latest, int count) {
  if (count == 0) {
    return latest;
  } else {
    return fibonacci(latest, previous + latest, count-1);
  }
}
```

The Design Process

We will cover the whole process of designing programs.



Careful use of design processes can save time and reduce frustration, even with the fairly small programs written in this course.

Themes of the course

- Design (the art of creation)
- Abstraction (finding commonality, ignore irrelevant details)
- Refinement (revising and improving initial ideas)
- Syntax (how to say things), expressiveness (how easy it is to say and understand), and semantics (the meaning of what is said)
- Communication

We will start by doing some simple calculations.

Vocabulary:

- In $g(x, y) = x - y$, x and y are called the **parameters** of g. In a **function application** such as $g(5, 3)$, 5 and 3 are the **arguments** for the parameters.
- We **evaluate** an expression such as $g(3, 1)$ by **substitution**. Thus $g(3, 1) = 3 - 1 = 2$.
- The function **consumes** 3 and 1, and **produces 2**.

Built-in functions

There are many built in functions, too many to list here.
In the menus: *Help→Racket Documentation*.
A few of particular interest:

(quotient n m) performs integer division (discarding the remainder). For example,

`(quotient 75 7)` ⇒ 10

(remainder n m) computes the remainder of integer division. For example,

`(remainder 75 7)` ⇒ 5

What is wrong with each of the following?

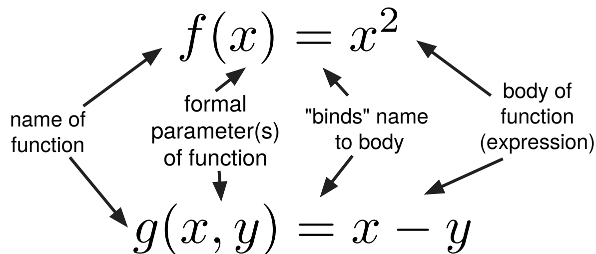
- `(* (5) 3)`
- `(+ (* 2 4)`
- `(5 * 14)`
- `(* + 3 5 2)`
- `(/ 25 0)`

Syntax refers to the ways we may express things. Racket syntax is very simple. Every expression is either a value such as a number, or of the form `(fname A B...)`, where `fname` is the name of a function, and `A` and `B` are expressions. (Details later.)

- `(* (5) 3)` contains a syntax error since `5` is not the name of a function.
- `(5 * 14)` has *the same* syntax error.
- `(+ (* 2 4)` contains a syntax error since the brackets don't match.

Semantics refers to the meaning of what we say. Semantic errors occur when an expression (which has correct syntax) cannot be reduced to a value by substitution rules.

- `(* + 3 5 2)` contains a semantic error since we cannot multiply "plus" with 3, 5, and 2.
- `(/ 25 0)` contains a semantic error since we cannot divide by zero.



A few 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.
- Different functions may use the same parameter name.
- Parameter order matters. $g(3, 1) = 3 - 1$ but $g(1, 3) = 1 - 3$.
- Calling a function creates a new value.

To translate $f(x) = x^2$ and $g(x, y) = x - y$ into Racket:

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

`define` is a special form. It looks like a Racket function but not all its arguments are evaluated. It **binds** a name to an expression. This expression may use the parameters which follow the name, along with other built-in and user-defined functions.

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

name: `g`; parameters: `x y`; body expression: `(- x y)`

Identifiers

Functions and parameters are named by identifiers, like `f`, `x`, `y`.

- Identifiers can contain letters, numbers, `-`, `_`, `.`, `?` and some other characters.
- Identifiers cannot contain space, brackets of any kind, or quotation marks like `` `'" ``

A few examples:

```
(define (g x y) (- x y))
(define (square-it! it!) (* it! it!))
(define (2remainder? n) (remainder n 2))
```

As with Mathematical functions:

- 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.
- Different functions may use the same parameter name; there is no problem with
  ```
  (define (f x) (* x x))
  (define (g x y) (- x y))
  ```
- Parameter order matters.
  ```
  (define (g x y) (- x y))
  ```
  and
  ```
  (define (g y x) (- x y))
  ```
  are not the same.

Constants in Racket

Racket also allows a special type called a constant:

```
(define k 3)
```

Now `k` gives 3.

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

The expression `(* k k)` is evaluated, and constant `p` takes the value 9.

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


There are a few built-in constants, including `pi` and `e`. Some programs might make their own constants, such as `interest-rate` or `step-size`.

Constants can make your code easier to understand and easier to change.

Note: what the CS 115 course notes calls “constants”, the textbook calls “variables”.

"Big red trucks drive quickly" is an English sentence with correct syntax and clear semantic interpretation.
"colorless green ideas sleep furiously" [1] has the same syntax, but no clear semantic interpretation.
"Students hate annoying professors" and "I saw her duck" both have ambiguous semantic interpretation; they have multiple possible meanings.
Computer languages are designed so every program has at most one semantic interpretation.

[1] from *Syntactic Structures* by Noam Chomsky, 1957.

Interpreting programs

Given these definitions:

```
(define foo 4)
(define (bar a b) (+ a a b))
```

What is the value of this expression?

```
(* foo (bar 5 (/ 8 foo)))
```

A model of Racket semantics

We wish to be able to predict the behaviour of any Racket program.

We can do this by viewing running a program as applying a set of **substitution rules**. Any expression which is not a definition we will simplify to a single **value**.

For example, consider

`(* 3 (+ 1 (+ (/ 6 2) 5)))`

Since the semantic interpretation of `(/ 6 2)` is `3`, we can simplify:

⇒ `(* 3 (+ 1 (+ 3 5)))`

> Complete the interpretation of `(* 3 (+ 1 (+ (/ 6 2) 5)))`

⇒ `(* 3 (+ 1 8))`

⇒ `(* 3 9)`

⇒ `27`

Now consider the following program:

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

The semantic interpretation of `(f 2)` is `(* 2 2)`, so simplify:

⇒ `(g (* 2 2) (g 3 1))`

Complete the interpretation of `(g (f 2) (g 3 1))`

⇒ `(g 4 (g 3 1))`

⇒ `(g 4 (- 3 1))`

⇒ `(g 4 2)`

⇒ `(- 4 2)`

⇒ 2

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

Recall from before:

"Every expression is either a value such as a number, or of the form `(fname A B...)`, where `fname` is the name of a function, and `A` and `B` are expressions."

> Major approach: to evaluate an expression such as `(fname A B)`, evaluate the arguments `A` and `B`, then apply the function to the resulting values.

For example, to evaluate `(+ (/ 6 2) 5)`, first we need to evaluate `(/ 6 2)`, which gives `3`. The other argument, `5`, is already evaluated. The expression becomes `(+ 3 5)`, so apply the `+` function to the values `3` and `5`, giving `8`.

Note: we do not evaluate definitions; we use definitions to evaluate expressions.

Let's all do it the same way

Evaluate arguments starting at the left.

For example, given `(* (+ 2 3) (+ 5 7))`, perform the substitution `(+ 2 3)` ⇒ `5` before the substitution `(+ 5 7)` ⇒ `12`.
(Note: this choice is arbitrary, and every choice will give the same final value. But if we all do it the same way it's easier to communicate what we are doing.)

Built-in function application use mathematical rules.

E.g. `(+ 3 5)` ⇒ `8`

Value no substitution needed.

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

```
(define x 3)
(* x (+ x 5))

⇒ (* 3 (+ x 5))
⇒ (* 3 (+ 3 5))
```

User-defined function application a function is 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`**. For example,

```
(define (foo a b) (+ a (- a b)))
(foo 4 3)

⇒ (+ 4 (- 4 3))
```

Note: each `vi` must be a value. To evaluate `(foo (+ 2 2) 3)`, *do not* substitute `(+ 2 2)` for `a`, to give `(+ (+ 2 2) (- (+ 2 2) 3))`.

Always remember to evaluate the arguments first.

What is the value?

```
(define x 4)
(define (f x) (* x x))
(f 3)
```

```
(define (huh? huh?) (+ huh? 2))
(huh? 4)
```

```
(define y 3)
(define (g x) (+ x y))
(g 5)
```

```
(define z 3)
(define (h z) (+ z z))
(h 7)
```

Applying simplification rules such as these allows us to predict what a program will do. This is called **tracing**.

Tracing allows you to determine if your code is semantically correct – that it does what is supposed to do.

If no rules can be applied but an expression cannot be further simplified, there is an error in the program. For example `(sqr 2 3)` cannot be simplified since `sqr` has only one parameter.

Be prepared to demonstrate tracing on exams.

The Stepper may not use the same tracing rules, even if it gives the same result. Write your own trace to ensure you understand your code.

Tracing practice

Trace the program:

```
(+ (remainder (- 10 2) (quotient 10 3)) (* 2 3))
⇒ (+ (remainder 8 (quotient 10 3)) (* 2 3))
⇒ (+ (remainder 8 3) (* 2 3))
⇒ (+ 2 (* 2 3))
⇒ (+ 2 6)
⇒ 8
```

Write a Racket function corresponding to

$$g(x, y) = x\sqrt{x} + y^2$$

```
(define (g x y) (+ (* x (sqrt x)) (* y y)))
```

Tracing practice

Trace the program: (Note: `(sqrt n)` computes $\sqrt{n}$ and `(sqr n)` computes n^2)

```
(define (disc a b c) (sqrt (- (sqr b) (* 4 (* a c)))))
(define (proot a b c) (/ (+ (- 0 b) (disc a b c)) (* 2 a)))
(proot 1 3 2)
```

```
⇒ (/ (+ (- 0 3) (disc 1 3 2)) (* 2 1))
⇒ (/ (+ -3 (disc 1 3 2)) (* 2 1))
⇒ (/ (+ -3 (sqrt (- (sqr 3) (* 4 (* 1 2))))) (* 2 1))
⇒ (/ (+ -3 (sqrt (- 9 (* 4 (* 1 2))))) (* 2 1))
⇒ (/ (+ -3 (sqrt (- 9 (* 4 2)))) (* 2 1))
⇒ (/ (+ -3 (sqrt (- 9 8))) (* 2 1))
⇒ (/ (+ -3 (sqrt 1)) (* 2 1))
⇒ (/ (+ -3 1) (* 2 1))
⇒ (/ -2 (* 2 1))
⇒ (/ -2 2)
⇒ -1
```

The official course notes have a list of goals at the end of each module. You should ensure you are able to do everything it lists.

Before we begin the next module, please

- Read *How to Design Programs*, sections 1-3
- Read the *Survival Guide* on assignment style & submission