Working with Atomic Data

Lecture 2
9 September 2019
Introduction
There’s much more to computer science than programming, but the experience of programming helps develop problem-solving skills – in particular, the ability to deal with complexity.
The most important concept in computer science is *abstraction*.

This is the process of treating something complex as if it were simpler, stripping away detail.

The most powerful abstracting notion we have is the simple act of *naming*, which allows any complex expression to be reduced to and referenced by a single name.
The inverse of abstraction is *synthesis* – the building of complexity from smaller pieces.
Every high-level programming language provides means for both abstraction and synthesis, including the one we’ll use this semester.
Racket is a recent variant of Scheme, a dialect of Lisp.

Lisp – the **LIS**t Processing language – is the second oldest high-level programming language, which was created in 1958 for artificial intelligence development.

Racket has a simple syntax that allows us to focus on *using* the language, rather than the language itself.
Programming is a creative activity that rewards the elegant use of abstraction techniques and good problem-solving style.

I hope you find programming in Racket to be both challenging and enjoyable, giving you the desire to learn other languages in the future and building a solid foundation for general problem-solving.
Expressions
In DrRacket, we’ve seen we can write expressions, e.g.,

\[(+ 3 4)\]

If we type an expression in the interactions window and type return or type it in the definitions window and click run, Racket will *evaluate* the expression to produce a value – in this case, 

7
Expressions can be more complicated, e.g., we could say

```lisp
> (+ 3 (* 2 3))
9

> (/ 12 (* 2 3))
2
```
Expressions have the forms

\langle \text{value} \rangle, \text{e.g., 7, or}

(\langle \text{primitive} \rangle \langle \text{expression} \rangle \ldots), \text{e.g., (+ 7 1)}
When we don’t want something to be evaluated as an expression, we can make it a comment by prefixing it with a semicolon (or, more often, two):

```
;; (+ 1 2) won’t be evaluated
```
Built-in *primitives* include basic math like

- for addition
- for subtraction
* is multiplication
/ is division

sqr squares a number
sqrt takes the square root
Recall the *Pythagorean Theorem* tells us that the area of a square whose sides are the same length as the hypotenuse of the triangle is equal to the sum of the area of the squares on the other two sides.

![Diagram of the Pythagorean Theorem](https://en.wikipedia.org/wiki/File:Pythagorean_theorem_04.png)

The area of square $S_1$ is the area of square $S_2$ + the area of square $S_3$
Written as an equation, \( a^2 + b^2 = c^2 \)

So, \( c = \sqrt{a^2 + b^2} = \sqrt{3^2 + 4^2} \)

What’s a BSL expression you can give to DrRacket that produces the value for \( c \)?
Written as an equation, \( a^2 + b^2 = c^2 \)

So, 
\[
\begin{align*}
    c &= \sqrt{a^2 + b^2} = \sqrt{3^2 + 4^2} \\
    &> (\sqrt{3^2 + 4^2}) \\
    &= 5
\end{align*}
\]
This is about the hardest math we’ll see in the whole course.

You can be a very good program designer without knowing a lot of math. For some domains, like computer graphics, you do need some advanced math, but that’s the same as needing to know biology to work in computational biology; it’s a domain skill rather than something really intrinsic to computational thinking or programming.
Evaluation
Read–Eval–Print Loop (REPL)

Racket does one thing, over and over:

- Read an expression typed in by the user.
- Evaluate the expression to obtain a value.
- Display the value of the expression.
Welcome to DrRacket.

> 1
1
> 2
2

User types the number “1” and hits the “Return” key.

Racket displays “>”, called the “prompt”.

Racket evaluates the expression “1” and displays its value.
Read–Eval–Print example

Welcome to DrRacket.

> (+ 8 7)
15

User types a more complex expression and hits the “Return” key.

> (- 15 8)
7

Racket evaluates the expression and displays its value.
How does Racket evaluate expressions to produce values?

E.g.,

\[ > (+ 2 (* 3 4) (- (+ 1 2) 3)) \]
\[ 14 \]
First, some terminology. This expression is a \textit{primitive call} because it starts with the name of a primitive operation.
Primitive call rule

To evaluate a primitive call,

1. Reduce operands to values

2. Apply the primitive to the values.
\((+ 2 (* 3 4)) (- (+ 1 2) 3)\)
\((+ 2 12) (- (+ 1 2) 3)\)
\((+ 2 12) (- 3 3)\)
\((+ 2 12) 0\)

14
Interpreting expressions as trees

\((\ast (\ast 8 7) 2)\)

Arrows indicate direction of data flow.
Interpreting expressions as trees

\((+ (+ (+ 3 3) 3) 3) 3)\)
Intuitively, evaluation moves from left to right, inside to outside.
As we keep going, we’ll see a few more evaluation rules, for expressions that aren’t primitive calls.
Strings and images
Strings

A *string* is written as any text – except for a double quote – written inside double quotes, e.g.,

"aardvark"

If you want a double quote inside a string, you need to tell Racket that it’s not the end of the string by *escaping* it with a backslash:

"I said \"good day, sir!\""
We can join strings together using the `string-append` primitive:

```scheme
> (string-append "Vassar" "College")
"VassarCollege"

> (string-append "Vassar" " " "College")
"Vassar College"
```
Note that different primitives take different types of data as arguments.

The mathematical operators we saw like $+$ expect numbers and will complain if you give them strings:

```
> (+ 1 "23")
😡
```
Another operation that works on strings include **string-length**, which does what it says on the tin:

```
> (string-length "quokka")
6
```
There's also the **substring** primitive, which lets us retrieve parts of a string:

```lisp
> (substring "quokka" 2 4)
"ok"
```

Why is it "ok"? We use 0-based indexing and substring returns up-to-but-not-including the second index.

```
"quokka"
012345
```
To use images as data, we require one of the teachpacks, so we start our definitions with this line:

```
(Require 2htrdp/image)
```

This says that we want to use image functions from the 2nd edition of *How to Design Programs*. 
Solid (i.e., filled in) red circle with radius of 10 pixels:

(circle 10 "solid" "red")

30-by-60 blue outline of a rectangle:

(rectangle 30 60 "outline" "blue")

We can also make images of text:

(text "hello" 24 "orange"
Just as we can combine numbers with arithmetic and strings by appending, we can combine images, e.g.,

(above (circle 10 "solid" "red")
 (circle 20 "solid" "yellow")
 (circle 30 "solid" "green"))
Just as we can combine numbers with arithmetic and strings by appending, we can combine images, e.g.,

```
(beside (circle 10 "solid" "red")
   (circle 20 "solid" "yellow")
   (circle 30 "solid" "green"))
```
Just as we can combine numbers with arithmetic and strings by appending, we can combine images, e.g.,

\[
\text{(overlay (circle 10 "solid" "red")}
\text{(circle 20 "solid" "yellow")}
\text{(circle 30 "solid" "green")})
\]
There are lots of primitives that apply to images. We won’t go over all of them in class; you can look them up as needed.

Two important ones to point out: \texttt{image-width} and \texttt{image-height}
Constant definitions
We give names to constants because it’s much harder to maintain code that has specific values repeated throughout.

When a value needs to change, we only want to change it in one place.
If we define constants

(require 2htdp/image)
(define WIDTH 400)
(define HEIGHT 600)

Now if we write

(* WIDTH HEIGHT)

it gets evaluated:

(* 400 HEIGHT)
(* 400 600)
240000
General form for defining a constant:

\((define \langle name \rangle \langle expression \rangle)\)
Evaluation rules for constant definitions

To evaluate a constant definition,

1. Evaluate the expression and record the resulting value as the value of the constant with the given name

To evaluate a defined constant name,

1. The value is the recorded name.
(require 2htdp/image)
(define WIDTH 400)
(define HEIGHT 600)
(define CAT 🐱

To add an image, copy it, e.g., this cat is copied from HtDP, §1.4
> (rotate -10 CAT)

> (rotate 10 CAT)
(define R\textsf{CAT} (rotate \text{-10} CAT))
(define L\textsf{CAT} (rotate 10 CAT))
Names are arbitrary

The following is silly, but legal:

Welcome to DrRacket.

> (define FIVE 6)

> FIVE

6

> (define SIX 5)

> SIX

5
Several constants may have the same value:

Welcome to DrRacket.

> (define SEVEN 7)
> SEVEN
7
> (define SEPT 7)
> SEPT
7
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Gregor Kiczales
Shriram Krishnamurthi