Atomic Data

3 September 2020
There’s much more to computer science than programming, but the experience of programming develops computational thinking skills.
For the rest of the semester we won’t be programming in Logo, no matter how much we like turtles.

Instead, we’ll be using dialects of Racket.
For the rest of the semester we won’t be programming in Logo, no matter how much we like turtles.

Instead, we’ll be using dialects of Racket.
Racket is a modern variant of *Scheme*, which is a dialect of *Lisp*.

Lisp – the LISt 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 on the language itself.
If you have experience with other programming languages, the way Racket works may be surprising, and programming in it may feel counterintuitive.
Writing expressions
In DrRacket, we can write *expressions*, e.g.,

\[
> (+ 3 4)
\]

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

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

\[
> (+ 3 (* 2 3)) \\
9 \\
> (/ 12 (* 2 3)) \\
2
\]
Racket expression notation

Put all operators at the front

Start every operation with an open parenthesis

Put a closing parenthesis after the last argument

Never add extra parentheses

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Racket</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 + 2</td>
<td>(+ 1 2)</td>
</tr>
<tr>
<td>4 + 2 × 3</td>
<td>(+ 4 (* 2 3))</td>
</tr>
<tr>
<td>(\cos(0) + 1)</td>
<td>(+ (cos 0) 1)</td>
</tr>
</tbody>
</table>
Built-in *primitives* we can apply to numbers include:

- Addition
- Subtraction
- Multiplication
- 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 other two sides.

$$S_1 = S_2 + S_3$$

*Figure from Wikipedia*
Written as an equation, $a^2 + b^2 = c^2$.

So, $c = \sqrt{a^2 + b^2} = \sqrt{3^2 + 4^2}$.

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

So, $c = \sqrt{a^2 + b^2} = \sqrt{3^2 + 4^2}$.

> `(sqrt (+ (sqr 3) (sqr 4)))`
5
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 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.
Evaluating expressions
Racket does the same thing, over and over:

*Read* an expression typed in by the user,

*Evaluate* the expression to obtain a value, and

*Print* the value of the expression.

This is called the *read–eval–print loop* (REPL).
Read–eval–print example

Welcome to DrRacket.
> 1
1
> 2
2
Read–eval–print example

Welcome to DrRacket.

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

1
1
> 2
2
Read–eval–print example

Welcome to DrRacket.

> 1
1
> 2
2

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

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

> 1

1

> 2

2

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

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

Racket evaluates the expression “1” and displays its value.
Read–eval–print example

Welcome to DrRacket.

> (+ 8 7)
15

> (- 15 8)
7

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

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

It’s a lot like basic arithmetic.
Arithmetic has fixed, pre-defined rules for *primitive operators*:

\[ 2 + 3 = 5 \]

\[ 4 \times 2 = 8 \]

\[ \cos(0) = 1 \]
Arithmetic has fixed, pre-defined rules for \textit{primitive operators}:

\[
\begin{align*}
2 + 3 & \rightarrow 5 \\
4 \times 2 & \rightarrow 8 \\
\cos(0) & \rightarrow 1
\end{align*}
\]
Arithmetic has fixed, pre-defined rules for *primitive operators*:

\[ 2 + 3 \rightarrow 5 \]
\[ 4 \times 2 \rightarrow 8 \]
\[ \cos(0) \rightarrow 1 \]

And it has rules for combining other rules:

Evaluate sub-expressions first:

\[ 4 \times (2 + 3) \rightarrow 4 \times 5 \rightarrow 20 \]
Arithmetic has fixed, pre-defined rules for *primitive operators*:

\[
2 + 3 \rightarrow 5 \\
4 \times 2 \rightarrow 8 \\
\cos(0) \rightarrow 1
\]

And it has rules for combining other rules:

Evaluate sub-expressions first:

\[
4 \times (2 + 3) \rightarrow 4 \times 5 \rightarrow 20
\]

Precedence determines subexpressions:

\[
4 + 2 \times 3 \rightarrow 4 + 6 \rightarrow 10
\]
This expression is a *primitive call* because it starts with the name of a primitive operation.

\[
(+ 2 (* 3 4) (- (+ 1 2) 3))
\]
To evaluate a primitive call,

Reduce operands to values

Apply the primitive to the values.
\[(+ \ 2 \ (* \ 3 \ 4) \ (- \ (+ \ 1 \ 2) \ 3))\]
\[
\rightarrow (+ \ 2 \ 12 \ (- \ (+ \ 1 \ 2) \ 3))
\]
\[
\rightarrow (+ \ 2 \ 12 \ (- \ 3 \ 3))
\]
\[
\rightarrow (+ \ 2 \ 12 \ 0)
\]
\[
\rightarrow 14
\]
Intuitively, evaluation moves from left to right, inside to outside.
Interpreting expressions as trees

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

Arrows indicate direction of data flow.
Interpreting expressions as trees

\((+ (+ (+ 3 3) 3) 3)\)
As we keep going, we’ll see a few more evaluation rules, for expressions that aren’t primitive calls.
When we don’t want something to be evaluated as an expression, we can make it a comment by prefixing it with one or more semicolons:

```plaintext
;; (+ 1 2) won’t be evaluated
```
Atomic data
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:

> `(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:

```scheme
> (+ 1 "23")
```

😡
Another operation that works on strings include **string-length**, which does what it says on the tin:

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

```scm
> (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.

```text
"quokka"
012345
Images
To use images as data, we require one of the textbook’s “teachpacks”, so we start our definitions with this line:

```scheme
(require 2htdp/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")

30×60 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.,

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

\[
\text{(overlay (circle 10 "solid" "red") (circle 20 "solid" "yellow") (circle 30 "solid" "green"))}
\]
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: **image-width** and **image-height**
Constant definitions
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 basic form of abstraction is *naming*, which allows any complex expression to be reduced to a single name.
> (define YEAR 2020)
> (define YEAR 2020)

special word to remember something
> (define YEAR 2020)

label (for a human)
> (define YEAR 2020)

value (the computer remembers)
> (define YEAR 2020)
> (define YEAR 2020)
> YEAR
> (define YEAR 2020)
> YEAR
2020
> (define YEAR 2020)
> YEAR
2020
> (define YEAR 2021)

year: this name was defined previously and cannot be re-defined
> `(define YEAR 2020)`
> YEAR
> 2020
> `(define YEAR 2021)`

*year: this name was defined previously and cannot be re-defined*

What we defined is a constant; it doesn’t change.
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.
> (define YEAR 2020)
> (define GRAD (+ YEAR 4))
> GRAD
> 2024
Acknowledgments

This lecture incorporates material from:


Gregor Kiczales, University of British Columbia

Marc Smith, Vassar College

Laney Strange, Northeastern University