CMPU 101 § 55

Computer Science I

Problem-Solving and Abstraction

Prof. Jonathan Gordon

Lecture 2

Introduction

There's much more to computer science than programming.

- However, until you understand how to communicate with a computer and how to teach the computer to solve problems, it's difficult to fathom more complicated computing topics.
- Even though computing technology has advanced tremendously, most communication with the computer still involves programming.

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, throwing away detail.
- The most powerful abstracting notion we have is *naming*, which allows any complex expression to be reduced to and referenced by a single name or symbol.

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.

Racket – previously known as Scheme – is a dialect of Lisp.

- Lisp – the **LIS**t **P**rocessing language – is the second oldest programming language.
- Lisp 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.

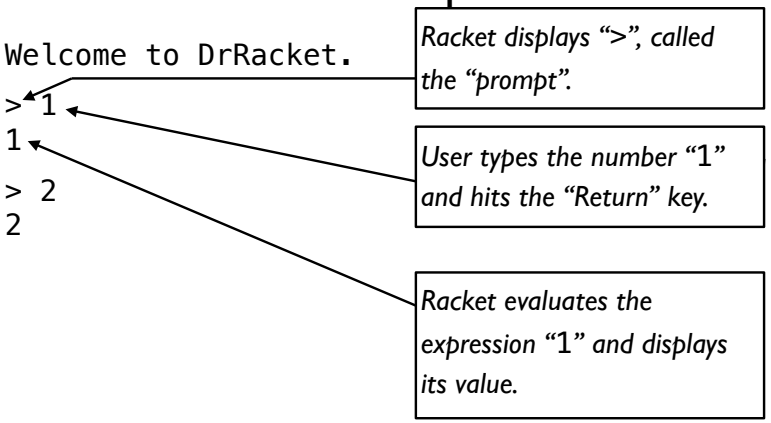
Evaluating expressions

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.

Read–Eval–Print example



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.

Kinds of expressions in Racket

Primitive expressions:

Constants, such as numbers: 1, 2, …

Variables, such as `name`, `age`,…

Composite expressions:

Expressions that are composed of other, smaller expressions.

Example: `(+ 8 7)`, which means 8 + 7.

Example: `(- 15 8)`, which means 15 − 8 .

Types of values in Racket

Numbers: 1, 2, 17, `3.14159`.

Symbols: `arthur`, `ford`, `marvin`.

Lists: `(arthur ford)`, `(marvin eddie)`.

Booleans: `#t`, `#f`.

Strings: `"Vassar College"`, `"To be or not to be?"`.

Procedures: `#<primitive:+>`, `#<primitive:*>`.

Composite expressions

Racket uses prefix notation – operators are at the front

Expressions are enclosed in parentheses

E.g.,

1 + 2 → `(+ 1 2)`

Examples of evaluating composite expressions:

```
Welcome to DrRacket.
> (+ 8 7)
15
> (- 15 8)
7
```

Examples of evaluating composite expressions:

```
Welcome to DrRacket.
> (* 8 7)
56
> (/ 56 8)
7
```

Translating infix expressions into prefix expressions

```
8   +   7
```

Move "+" in front of "8" and "7".

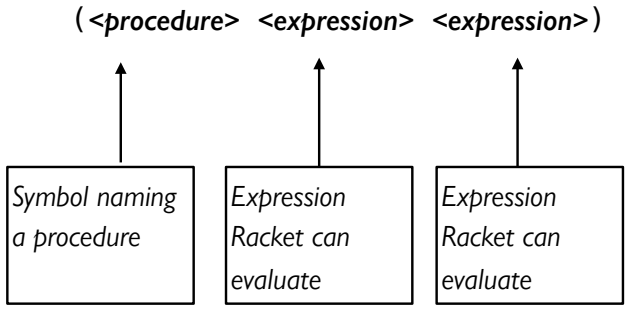
```
+   8   7
```

Add parentheses.

```
(+   8   7)
```

Syntax of composite expressions

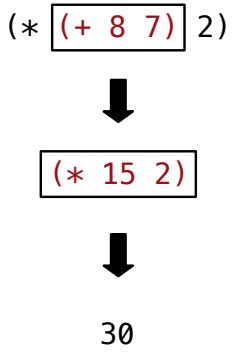
Procedures can accept zero, one, two, or more expressions as data to be used in computation. These expressions are called *parameters* or *arguments*.



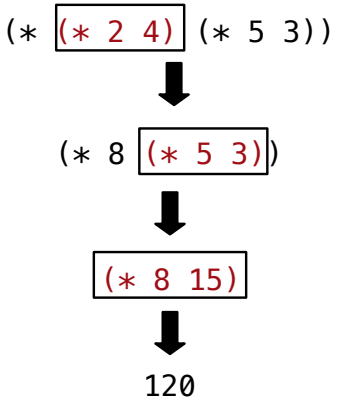
Expressions may be used inside larger expressions:

```
Welcome to DrRacket.
> (* (+ 8 7) 2)
30
> (* (* 2 4) (* 5 3))
120
> (+ (+ (+ 3 3) 3) 3)
12
```

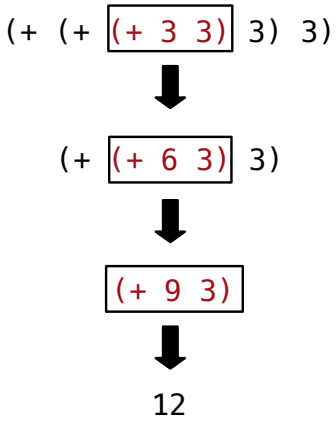
Steps in evaluating complex expressions:



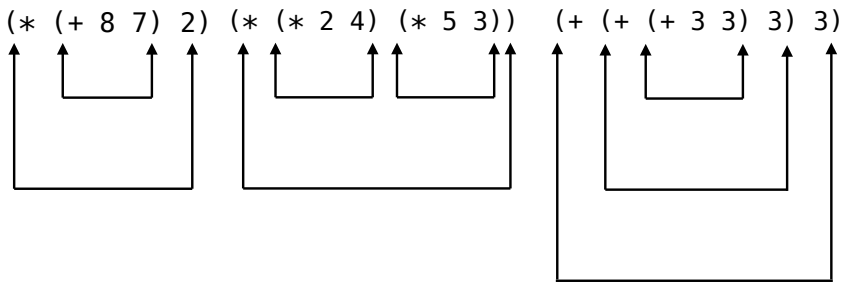
Steps in evaluating complex expressions:



Steps in evaluating complex expressions:



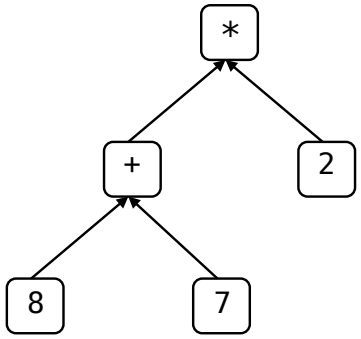
Parentheses must be balanced:



Interpreting expressions as trees

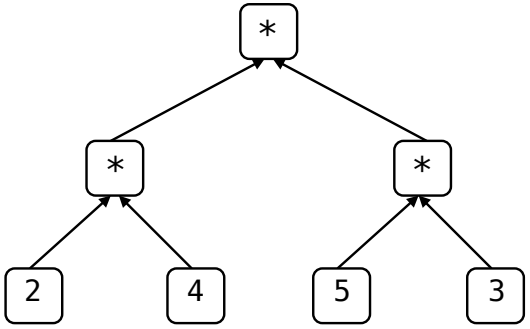
(* (+ 8 7) 2)

Arrows indicate direction of data flow.



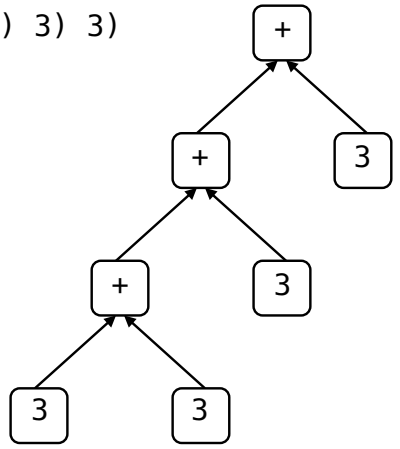
Interpreting expressions as trees

(* (* 2 4) (* 5 3))



Interpreting expressions as trees

(+ (+ (+ 3 3) 3) 3)



Defining variables

Programming languages have *keywords*, which are special symbols used by the language.

- In Racket, these include `and`, `cond`, `define`, `else`, `if`, `lambda`, `let`, `or`, and `quote`.

When the first element of a non-empty list in Racket is a keyword symbol, then the list is a *special form*, e.g.,

```
(define x 3)
(quote (3 4 5))
(if condition <then-clause> <else-clause>)
(let ((x 4)) (+ x 8))
```

The evaluation of special forms is different than for normal expressions.

- For each type of special form, there's a specific rule for how it is evaluated, determined by the keyword it uses.
- We'll see about a dozen kinds of special forms over the semester, with their own evaluation rules, but you'll use them so often that their evaluation will become second nature.

Definition of a variable

The *global environment* is a table that holds the values assigned to variables.

The `define` special form is the way we add entries to the global environment, so we can use them later.

- We can use `define` to add names for constant values and – as we'll see next class – functions.

Examples of defining variables

```
Welcome to DrRacket.
> (define one 1)
> one
1
> (define two 2)
> two
2
```

User tells Racket that 1 is the value of the variable "one".

User asks Racket to evaluate the variable "one".

Racket displays the value of the variable "one".

Variable names are arbitrary

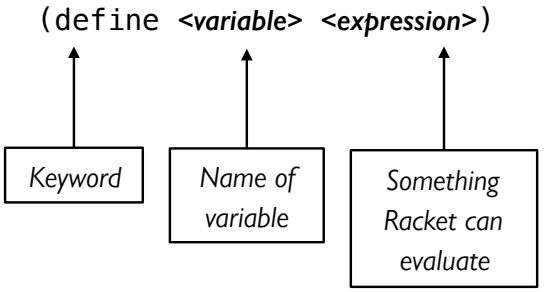
The following is silly, but legal:

```
Welcome to DrRacket.
> (define five 6)
> five
6
> (define six 5)
> six
5
```

Several variables may have the same value:

```
Welcome to DrRacket.
> (define seven 7)
> seven
7
> (define sept 7)
> sept
7
```

Syntax of variable definitions



Evaluation of **`define`** special forms

`(define` <***variable***> <***expression***>`)`

`(define graduation (+ 2019 4))`

Racket does **not** evaluate the keyword `define`.

Racket does **not** evaluate the **<*variable*>**.

Racket **does** evaluate the **<*expression*>**.

```
Welcome to DrRacket.
> (define w1 0.25)
```

Global Environment	
w1	0.25

```
Welcome to DrRacket.
> (define w1 0.25)
> (define w2 (+ 0.75 w1))
```

Global Environment	
w1	0.25
w2	1.0

```
Welcome to DrRacket.
> (define w1 0.25)
> (define w2 (+ 0.75 w1))
> (define w1 0)
```

Global Environment	
w1	0
w2	1.0

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.

Once a variable is defined, it may be used in any expression:

```
Welcome to DrRacket.
> (define year 2019)
> (+ year 4)
2023
```

Any expression may be used in the definition of a variable:

```
Welcome to DrRacket.
> (define year 2019)
> (define graduation (+ year 4))
> graduation
2023
```

```
Welcome to DrRacket.
> (define john-age 8)
> (define jane-age 7)
> (define pat-age 2)
```

Evaluating `(+ john-age jane-age)`

Evaluate + to get `#<primitive:+>`

Evaluate `john-age` to get 8

Evaluate `jane-age` to get 7

Apply `#<primitive:+>` to 8 and 7 to get 15

```
Welcome to DrRacket.
> +
#<primitive:+>
> john-age
8
> jane-age
7
> (+ john-age jane-age)
15
```

Defining variables whose values are symbols

```
Welcome to DrRacket.
> (define director ron)
reference to an identifier before its
definition: ron
```

What went wrong?

Racket tried to evaluate the variable "`ron`".

We had not previously defined "`ron`".

How can we talk about the ***symbol*** "`ron`", rather than the ***variable*** "`ron`"?

The **`quote`** special form

Inhibits the normal evaluation of its argument.

General form: `(quote` ***<expression>***`)`, e.g.,

```
(quote ron)
(quote (1 2 3)
```

Shorthand form: **`'<expression>`**, e.g.,

```
'ron
```

Even though the shorthand looks very different, it is the same expression.

Defining variables whose values are symbols

```
Welcome to DrRacket.
> (define director 'ron)
> director
'ron
> (define deputy-director 'leslie)
> deputy-director
'leslie
```

Lists of data

Constructing lists in Racket

```
Welcome to DrRacket.
> '()
'()
> '(harry)
'(harry)
> '(harry hermione)
'(harry hermione)
> '(harry hermione ron)
'(harry hermione ron)
```

Constructing lists in Racket

```
> '()
'()
> (cons 'harry '())
'(harry)
> (cons 'harry (cons 'hermione '()))
'(harry hermione)
> (cons 'harry
        (cons 'hermione
              (cons 'ron '())))
'(harry hermione ron)
```

The cons procedure

`(cons` ***<item>*** ***<list>***`)`

Constructs lists of values.

First argument can be anything.

Second argument is normally a list.

`cons` will construct a new list by inserting ***<item>*** at the beginning of ***<list>***.

The empty list

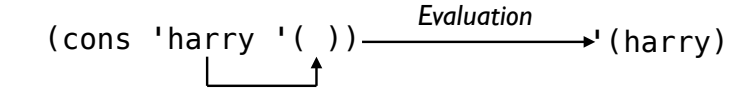
A list with nothing on it.

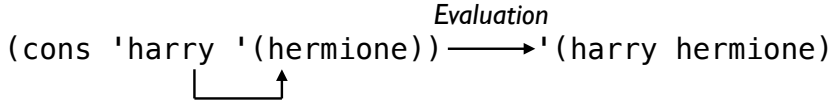
User types `'()` to Racket.

Racket evaluates `'()` to get `'()`.

Racket displays `'()` as the value of `'()`.

An Illustration of **cons**





Constructing lists in Racket

```
> (define first-couple
    (cons 'barack (cons 'michelle '())))
> first-couple
'(barack michelle)
> (define second-couple
    (cons 'joe (cons 'jill '())))
> second-couple
'(joe jill)
> (define leaders
    (cons first-couple
          (cons second-couple '())))
> leaders
'((barack michelle) (joe jill))
```

… lists to be continued next time!

Acknowledgments

This lecture incorporates material from:

Tom Ellman

Luke Hunsberger