CS 135
September 10, 2019
Programming language design
Values, expressions, & functions
The DrRacket environment
Programming in DrRacket
Defining functions

Functions, continued
Functions in mathematics – Ambiguity

Definitions: $f(x) = x^2$, $g(x,y) = x + y$

There are many mathematically valid substitutions:

• $g(g(1,3), f(3)) = g(1 + 3, f(3))$
• $g(g(1,3), f(3)) =$
Functions in mathematics – Ambiguity

Definitions: \( f(x) = x^2 \), \( g(x, y) = x + y \)

There are many mathematically valid substitutions:

• \( g(g(1,3), f(3)) = g(1 + 3, f(3)) \)
• \( g(g(1,3), f(3)) = g(g(1,3), 3^2) \)
Functions in mathematics – Ambiguity

Definitions: \( f(x) = x^2, g(x, y) = x + y \)

There are many mathematically valid substitutions:

- \( g(g(1,3), f(3)) = g(1 + 3, f(3)) \)
- \( g(g(1,3), f(3)) = g(g(1,3), 3^2) \)
- \( g(g(1,3), f(3)) = g(1,3) + f(3) \)
Functions in mathematics – Ambiguity

Definitions: $f(x) = x^2$, $g(x, y) = x + y$

There are many mathematically valid substitutions:

- $g(g(1,3), f(3)) = g(1 + 3, f(3))$
- $g(g(1,3), f(3)) = g(g(1,3), 3^2)$
- $g(g(1,3), f(3)) = g(1,3) + f(3)$

We would like a canonical form for two reasons:

- Easier for us to think about
- When we extend this idea to programming, we will find cases where different orderings result in different values
Functions in mathematics – Canonical substitutions

Two rules:
1. Functions are applied to values
2. When there is a choice of possible substitutions, always take the leftmost choice.

Now, for any expression:
• there is at most one choice of substitution;
• the computed final result is the same as for other choices.
Functions in mathematics – Parentheses for ordering

In arithmetic expressions, we often place operators between their operands.

• Example: $3 - 2 + 4/5$.

We have some rules (division before addition, left to right) to specify order of operation.

Sometimes these do not suffice, and parentheses are required. Example:

$6 - 4/5 + 7$ vs.

$(6 - 4)/(5 + 7)$
If we treat infix operators (+, −, etc.) like functions, we do not need parentheses to specify order of operations:
Functions in mathematics – Parentheses for ordering

If we treat infix operators (+, −, etc.) like functions, we do not need parentheses to specify order of operations:

• Example:
  • 3 − 2 becomes
  • −(3,2)

• Example:
  • (6 − 4)/(5 + 7) becomes
  • /(−(6,4), +(5,7))
If we treat infix operators (+, −, etc.) like functions, we do not need parentheses to specify order of operations:

• Example:
  • 3 − 2 becomes
  • −(3,2)

• Example:
  • (6 − 4)/(5 + 7) becomes
  • /(-(6,4), +(5,7))
Functions in mathematics – Parentheses for ordering

If we treat infix operators (+, −, etc.) like functions, we do not need parentheses to specify order of operations:

• Example:
  • 3 − 2 becomes
  • −(3,2)

• Example:
  • (6 − 4)/(5 + 7) becomes
  • /(−(6,4), +(5,7))

The substitution rules we developed for functions now work uniformly for functions and operators. Parentheses now have only one use: function application.
Functions in mathematics – Parentheses for ordering

Racket writes its functions slightly differently: the function name moves inside the parentheses, and the commas are changed to spaces.

• Example: \( g(1, 3) \) becomes \((g \ 1 \ 3)\)

• Example: \( (6 - 4)/(5 + 7) \) becomes \((/ \ (\ - \ 6 \ 4) \ (+ \ 5 \ 7))\) These are valid Racket expressions (once \( g \) is defined).

Functions and mathematical operations are treated exactly the same way in Racket.
Expressions in Racket

\[ 3 - 2 + \frac{4}{5} \text{ becomes } (+ (- 3 2) (/ 4 5)) \]
\[ (6 - 4)(3 + 2) \text{ becomes } (* (- 6 4) (+ 3 2)) \]

Extra parentheses are harmless in arithmetic expressions. They are harmful in Racket.

Only use parentheses when necessary (to signal a function application or some other Racket syntax).
Expressions in Racket – Evaluation

We use a process of substitution, just as with our mathematical expressions.

Each step is indicated using the ‘yields’ symbol =>.

\[
\begin{align*}
\ast (\ast (- 6 4) (+ 3 2)) & \\
\Rightarrow (\ast 2 (+ 3 2)) & \\
\Rightarrow (\ast 2 5) & \\
\Rightarrow 10 & 
\end{align*}
\]
The DrRacket environment
Basic features

Designed for education, powerful enough for “real” use.

Sequence of language levels keyed to textbook.

Includes good development tools.

Two windows: Interactions (now) and Definitions (later)

Interactions window: a read-evaluate-print loop (REPL)
Setting the language

CS 135 will progress through the Teaching Languages starting with Beginning Student.

1. Under the Language tab, select Choose Language ...
2. Select Beginning Student under Teaching Languages
3. Click the Show Details button in the bottom left
4. Under Constant Style, select true false empty

Remember to follow steps 3 and 4 every time you change the language.
Setting the language

CS 135 will progress through the Teaching Languages starting with Beginning Student.

1. Under the Language tab, select Choose Language ...
2. Select **Beginning Student** under Teaching Languages
3. Click the Show Details button in the bottom left
4. Under Constant Style, select **true false empty**

Remember to follow steps 3 and 4 each time you change the language.
Constant style

In the DrRacket documentation, you will see #true, #false, '() instead of true, false, empty, respectively.

In CS 135 exams and stepper questions you must use true, false, empty.
Programming in DrRacket
Numbers in Racket

Integers in Racket are unbounded.

Rational numbers are represented exactly.

Expressions whose values are not rational numbers are flagged as being inexact.

Example: (sqrt 2) evaluates to #i1.414213562370951.

We will not use inexact numbers much (if at all).
Expressions in Racket

Racket has many built-in functions which can be used in expressions:

• Arithmetic operators: +, -, *, /
• Constants: e, pi
• Functions: (abs x), (max x y ...), (ceiling x), (expt x y),
  (exp x), (cos x), ...

Look in DrRacket’s “Help Desk”. The web page that opens has many sections. The most helpful is under Teaching, then “How to Design Programs Languages”, section 1.5.
Racket expressions causing errors

What is wrong with each of the following?

1. \((5 \ast 14)\)
2. \((\ast (5) 3)\)
3. \((+ (\ast 2 4)\)
4. \((\ast + 3 5 2)\)
5. \((/ 25 0)\)
Racket expressions causing errors

What is wrong with each of the following?

1. \((5 \times 14)\) # not in prefix notation
2. \((\times (5) 3)\) # too many brackets
3. \((+ (* 2 4))\) # second parameter for + missing
4. \((* + 3 5 2)\) # not enough brackets
5. \((/ 25 0)\) # division by 0
Defining functions
Defining functions – Mathematics

\[ g(x, y) = x + y \]

1. Function name
2. Formal parameter(s)
3. Binds name to body
4. Body of function (expression)
Defining functions – Racket

\[ g(x, y) = x + y \]

1. Function name
2. Formal parameter(s)
3. Binds name to body
4. Body of function (expression)
Defining functions – Racket

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

```
(define (f x) (sqr x))
(define (g x y) (+ x y))
```

`define` is a special form (looks like a Racket function, but not all of its arguments are evaluated).

It binds a name to an expression (which uses the parameters that follow the name).
Defining functions – Racket

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.

\[
\text{(define } (\text{func } x \ y) (\text{sqrt } (+ (\text{sqr } x) (\text{sqr } y))))
\]
Applying user-defined functions

An application of a user-defined function substitutes arguments for the corresponding parameters throughout the definition’s expression.

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

The substitution for \((\text{g } 3 \ 5)\) would be \((+ \ 3 \ 5)\).
Applying user-defined functions

When faced with choices of substitutions, we use the same rules defined earlier: apply functions only when all arguments are simple values; when you have a choice, take the leftmost one.

\[
g(g(1,3), f(3)) \\
= g(1 + 3, f(3)) \\
= g(4, f(3)) \\
= g(4, 3^2) \\
= g(4, 9) \\
= 4 + 9 \\
= 13
\]

\[
(g \ (g \ 1 \ 3) \ (f \ 3)) \\
\Rightarrow (g \ (+ \ 1 \ 3) \ (f \ 3)) \\
\Rightarrow (g \ 4 \ (f \ 3)) \\
\Rightarrow (g \ 4 \ (sqr \ 3)) \\
\Rightarrow (g \ 4 \ 9) \\
\Rightarrow (+ \ 4 \ 9) \\
\Rightarrow 13
\]
Defining constants

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

(\texttt{define } k \ 3)
(\texttt{define } p (\texttt{sqr} \ k))

The effect of (\texttt{define } k \ 3) is to bind the name $k$ to the value 3.

In (\texttt{define } p (\texttt{sqr} \ k)), the expression (\texttt{sqr} \ k) is first evaluated to give 9, and then $p$ is bound to that value.
Advantages of constants

Constants can

• give meaningful names to useful values (e.g. interest-rate, passing-grade, or starting-salary).

• reduce typing and errors when such values need to be changed

• make programs easier to understand

Constants can be used in any expression, including the body of function definitions.

Constants are sometimes called variables, but their values cannot be changed (until CS 136).
Scoping

(define x 3)

(define (g a b) (+ a b x))

... Generally, identifiers, such as constant and function names, have to be unique.
Scoping

 Generally, identifiers, such as constant and function names, have to be unique.
Scoping

Generally, identifiers, such as constant and function names, have to be unique.

```
(define x 3)
(define (g a b) (+ a b x))
(define (f x y) (- x y))
(f 5 1)
```
Scoping

Generally, identifiers, such as constant and function names, have to be unique.
Scoping

Identifiers, such as constant and function names, have to be unique within their scope.
Scoping

There are two scopes:
- Global scope
- Function scope

The smallest enclosing scope has priority.

```
(define x 3)
(define (g a b) (+ a b x))
(define (f x y) (- x y))
(f 5 1)
```
Scoping

There are two scopes:

- **Global scope**
- **Function scope**

The smallest enclosing scope has priority.

Racket tries to bind identifiers in a function body (here: a, b, x, y) to a value in the smallest scope first (here: parameters in the function scope).
Scoping

There are two scopes:

- **Global scope**
- **Function scope**

The smallest enclosing scope has priority.

If this is not successful, Racket then tries binding to a constant in the **global scope**.
Scoping – Parameter names

Each parameter name has meaning only within the body of its function.

\[
(\text{define } (f \ x \ y) \ (+ \ x \ y))
\]

\[
(\text{define } (g \ x \ z) \ (* \ x \ z))
\]

The two uses of \(x\) are independent.
Scoping – Parameter names

Each parameter name has meaning only within the body of its function.

\[(\text{define} \ (f \ x \ y) \ (+ \ x \ y))\]

\[(\text{define} \ (g \ x \ z) \ (* \ x \ z))\]

The two uses of x are independent.

Additionally, the following two function definitions implement the same functionality:

\[(\text{define} \ (f \ x \ y) \ (+ \ x \ y))\]
\[(\text{define} \ (f \ a \ b) \ (+ \ a \ b))\]
Scoping

The scope of an identifier indicates where it has effect within the program.

The smallest enclosing scope has priority.

A program cannot have duplicate identifiers within the same scope.
Scoping – Exploring bindings in DrRacket

(define x 3)
(define (g a b) (+ a b x))
(define (f x y) (- x y))
(g 5 1) ; 9
(f 5 1) ; 4
(f 5 x) ; 2
A Racket program is a sequence of definitions and expressions.

The expressions are evaluated, using substitution, to produce values.

Expressions may also make use of special forms (e.g. `define`), which look like functions, but don’t necessarily evaluate all their arguments.
DrRacket – Windows

Definitions window accumulates definitions and expressions
DrRacket – Windows

Definitions window accumulates definitions and expressions

Run button loads contents into Interactions window
DrRacket – Windows

Definitions window accumulates definitions and expressions

Run button loads contents into Interactions window

Provides a stepper to let one evaluate expressions step-by-step
DrRacket – Windows

Definitions window accumulates definitions and expressions

Run button loads contents into Interactions window

Provides a stepper to let one evaluate expressions step-by-step

Features: error highlighting, subexpression highlighting, syntax checking
Functions – End of module
Goals of this module

• You should understand the basic syntax of Racket, how to form expressions properly, and what DrRacket might do when given an expression causing an error.

• You should be comfortable with these terms: function, parameter, application, argument, constant, expression.

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

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