02: Functions

A computer program is a set of instructions to complete a particular task.

Many tasks are mathematical: the computation of certain mathematical values. This will be the primary direction we move in this course.

Many mathematical questions we can answer by hand. For example:

Ex. 1

How many natural numbers divide 12 evenly?

!

How many natural numbers divide $5,218,303$ evenly?

We give computers instructions using a **programming language**. Programming languages fall into "families" with common characteristics. Two such families are:

Imperative: based on frequent changes to data

- Examples: machine language, Java, C++, Turing, Visual Basic, Python

Functional: based on the computation of new values rather than the transformation of old ones.

- Examples: Excel formulas, LISP, ML, Haskell, Erlang, F#, Mathematica, XSLT, Clojure.

CS135 uses the language Racket, a member of the functional family of languages.

Designers of programming languages must solve three problems (illustrated here with English sentences):

1. **Syntax**: The way we're allowed to say things
   "?is This Sentence Syntactically Correct"
2. **Semantics**: What the program means
   "Trombones fly hungrily."
3. **Ambiguity**: Valid programs have exactly one meaning
   "Sally was given a book by Joyce."

English rules on these issues are pretty lax. For a programming language, we need rules that *always* avoid these problems.

Syntax and ambiguity can be solved with grammars, a topic covered in more depth in CS230, CS241, CS360, and CS444.

Racket allows us to easily develop a **semantic model** to specify the meaning of our programs using **substitution rules**. The first three rules will be developed in this module.

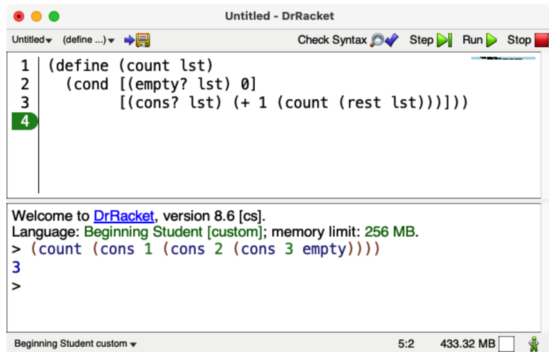
Other reasons to use Racket include:

- closely connected to mathematics
- functional languages are easier to design and reason about
- minimal but powerful syntax
- small toolbox with ability to construct additional required tools
- interactive evaluator
- graduated set of teaching languages
- levels the playing field with those who have programmed before

CS116 and CS136 use imperative programming languages. Functional and imperative share many concepts but also require you to think differently about your programs. Having experience in both is a good thing!

The DrRacket environment

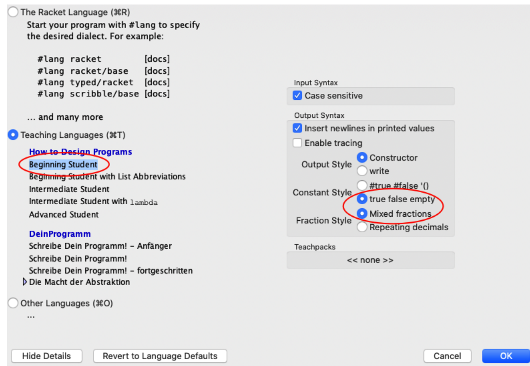
- Designed for education
- Sequence of language levels
- Two windows:
  - Definitions (top) used for writing programs
  - Interations (bottom) used for testing, experimenting



CS135 will progress through the Teaching Languages starting with *Beginning Student*.

Follow steps 3 - 5 each time you change the language.

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*
5. Under *Fraction Style*, select *Mixed fractions*



Values are numbers or other mathematical objects.
Examples: 5, $4/9$, π.

Expressions combine values with operators and functions.
Examples: $5 + 2$, $\sin(2\pi)$, $\frac{\sqrt{2}}{100\pi}$.

Functions generalize similar expressions.
Example:
$3^2 + 4(3) + 2$
$6^2 + 4(6) + 2$
$7^2 + 4(7) + 2$

are generalized by the function
$f(x) = x^2 + 4x + 2$.

- Integers in Racket are unbounded.
- Rational numbers are represented exactly: 2, $3\frac{1}{7}$
- Expressions whose values are not rational numbers are flagged as being **inexact**:
  `(sqrt 2)` ⇒ `#i1.414213562370951`.
  We will not use inexact numbers much.

```
Welcome to DrRacket, version 7.3 [3m].
Language: Beginning Student [custom];
memory limit: 128 MB.
> (expt 2 500)
327339060789614187001318969682759915221664204604306478948329136809613379640467455488327009232590415715088668412756007100921725654588539305332852758937 6
> (/ 3 5.55)
20
--
37
> (sqrt 5)
#i2.23606797749979
>
```

In time, we will add other kinds of values: symbols, Booleans, strings, etc.

Function definitions:

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

These definitions consist of:

- the name of the function (e.g. g)
- its **parameters** (e.g. x, y)
- an algebraic expression using the parameters as placeholders for values to be supplied in the future

Function definitions: $f(x) = x^2$
$g(x, y) = x + y$
$h(x) = \frac{x}{\sqrt{x}}$

An **application** of a function supplies **arguments** for the **parameters**, which are substituted into the algebraic expression.

Example: $g(1, 3) = 1 + 3 = 4$

An argument is substituted each time the associated parameter is used:

Example: $h(4) = \frac{4}{\sqrt{4}} = 2$

The arguments supplied may themselves be applications.

Example: $g(g(1, 3), f(3))$

Function definitions: $f(x) = x^2$
$g(x, y) = x + y$
$h(x) = \frac{x}{\sqrt{x}}$

We **evaluate** each of the arguments to yield values.

Evaluation by **substitution**:

$g(g(1, 3), f(3)) =$

$g(1 + 3, f(3)) =$

$g(4, f(3)) =$

$g(4, 3^2) =$

$g(4, 9) = 4 + 9 = 13$

Many possible substitutions

Function definitions:

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

There are many mathematically valid substitutions:

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

Having many different valid substitutions will cause trouble when we extend this to programs. So, we will:

- Apply functions only to values (expressions simplified first)
- When there is a choice of possible substitutions, always take the **leftmost** choice.

> The use of parentheses: function application

There are two uses of parentheses in our usual mathematical notation. We've just seen one of them: function application.

The parentheses identify the arguments the function is applied to.

$f(3)$

$g(1, 2)$

The second use of parentheses is to specify 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: $(6 - 4) \ / \ (5 + 7)$.

https://www.xkcd.com/992/

> The use of parentheses: harmonization

If we treat **infix operators** ($+$, $-$, etc.) like functions, we don't 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 process now works uniformly for functions and operators.

Parentheses now have only one use: function application.

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: $g(g(1,3), f(3))$ becomes `(g (g 1 3) (f 3))`

These are valid Racket expressions (once `g` and `f` are defined).

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

Example: $(6 - 4) \,/\, (5 + 7)$ becomes `(/ (- 6 4) (+ 5 7))`

Example: $3 - 2 + 4 \,/\, 5$ becomes `(+ (- 3 2) (/ 4 5))`

Racket supports fractions. $3 - 2 + 4 \ / \ 5$ can be written two ways:

- `(+ (- 3 2) (/ 4 5))`
- `(+ (- 3 2) 4/5)`

Extra parentheses are harmless in arithmetic expressions. Example: $(1 + (2 + 3))$

They are harmful in Racket. Example: `(+ (1 (+ 2 3)))` (invalid!)

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

Ex. 2

Transform each mathematical expression into an Racket expression. Enter them in DrRacket's interactions pane to check your work.

$2 + 3$ $\qquad$ 2×3 $\qquad$ $44 - 2$

$3 \times 4 + 2$ $\qquad$ $\dfrac{2 + 4}{5 - 1}$ $\qquad$ $3(1 + (6 / 2 + 5))$

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

Each step is indicated using the ‘yields’ symbol $\Rightarrow$.

```
(* (- 6 4) (+ 3 2)) ⇒
(* 2 (+ 3 2)) ⇒
(* 2 5) ⇒
10
```

The substitution process repeatedly simplifies the program. At each step, the result is a valid (but simpler) Racket program. It eventually simplifies to a value.

A **substitution step** finds the **leftmost subexpression eligible for rewriting**, and rewrites it by the rules we will describe.

Rule 1: Application of built-in functions

This is our first of the substitution rules, which form our semantic model.

We reuse the rules for the arithmetic expressions we are familiar with to substitute the appropriate value for expressions like `(+ 3 5)` and `(expt 2 10)`.

```
(+ 3 5) ⇒ 8
(expt 2 10) ⇒ 1024
```

Formally, the substitution rule is:

```
(f v1 ... vn) ⇒ v
```

where `f` is a built-in function, `v1` ... `vn` are values, and `v` is the value of $f(v_1, \ldots, v_n)$.

Note the two uses of an **ellipsis** (. . .). What does it mean?

For built-in functions `f` with one parameter, the rule is:
`(f v1)` $\Rightarrow$ `v` where `v` is the value of $f(v_1)$

For built-in functions `f` with two parameters, the rule is:
`(f v1 v2)` $\Rightarrow$ `v` where `v` is the value of $f(v_1, v_2)$

For built-in functions `f` with three parameters, the rule is:
`(f v1 v2 v3)` $\Rightarrow$ `v` where `v` is the value of $f(v_1, v_2, v_3)$

We can't just keep writing down rules forever, so we use ellipses to show a *pattern*:
`(f v1 ... vn)` $\Rightarrow$ `v` where `v` is the value of $f(v_1, \ldots, v_n)$.

What is wrong with each of the following?

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

Syntax error: An error discovered when reading an expression.
Run-time error: An error discovered when evaluating an expression.

DrRacket Documentation

Racket has many built in functions, too many to list here. To learn about the built-in functions, we need to read the documentation.

Ex. 3 In DrRacket, select *Help* → *Racket Documentation*

This brings up the web browser, like this →

Ex. 4 Scroll down to *Teaching* → *How to Design Programs Languages*, then *Beginning Student*.

file:///Applications/Racket%20v8.4.0.8/doc/index.ht

OAT:local EMod: Local CS135:local Computing Home Personal References Teaching UW PR CS349 AppDev

M02: Functions :: CS135 overflow-wrap - CSS: Cascading Style Shee... Racket Documentation

...search manuals...

Racket Documentation
Search Manuals

License
Acknowledgements
Release Notes

Report a Bug

v.8.4.0.8

Racket Documentation

This is an installation-specific listing. Running `raco docs` (or `Racket Documentation` on Windows or Mac OS) may open a different page with local and user-specific documentation, including documentation for installed packages.

Getting Started
Racket Cheat Sheet

Tutorials
Quick: An Introduction to Racket with Pictures
Continue: Web Applications in Racket
More: Systems Programming with Racket

Racket Language and Core Libraries
The Racket Guide
The Racket Reference

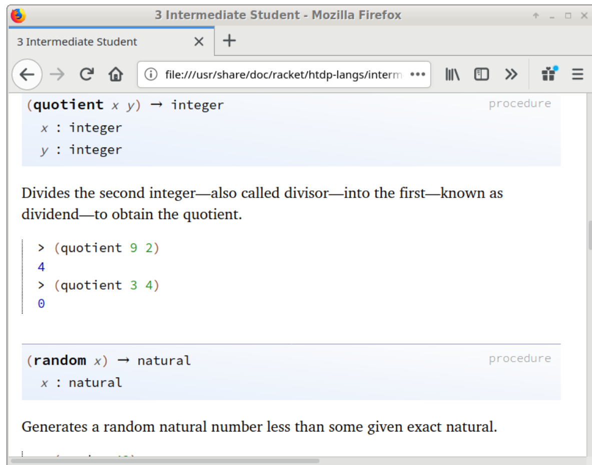
Package Management in Racket
The Racket Drawing Toolkit
The Racket Graphical Interface Toolkit
The Racket Foreign Interface
Scribble: The Racket Documentation Tool
DrRacket: The Racket Programming Environment
`raco`: Racket Command-Line Tools

How to Program Racket: a Style Guide

Teaching
How to Design Programs
How to Design Programs Languages
How to Design Programs Teachpacks

Finally we see information about the functions we are interested in: ⟶

Bookmark this page in your browser so you can find it quickly and easily.



Ex. 5

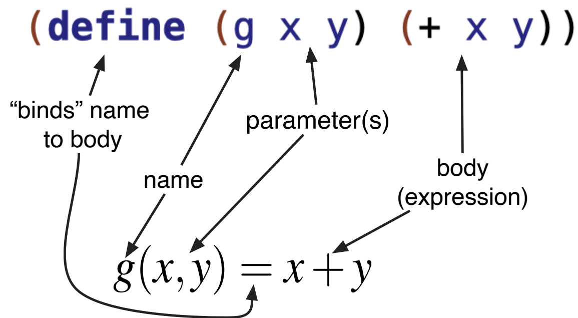
Become more comfortable with the documentation by looking up each of the following functions:

`quotient` `remainder` `expt` `gcd`

A function definition consists of:

- a **name** for the function,
- a list of **parameters**,
- a single **body** expression.

(Racket definition on top; math on the bottom.)



The body expression typically uses the parameters together with other built-in and user-defined functions.

Examples:

Math	Racket
$f(x) = x^2$	`(define (f x) (sqr x))`
$g(x, y) = x + y$	`(define (g x y) (+ x y))`
$area(r) = \pi r^2$	`(define (area r) (* pi (sqr r)))`

`define` is a **special form** (it 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).

Ex. 6

In DrRacket's definitions frame (the top one), use **`define`** to create a function `(add-twice a b)` that calculates $a + 2b$.

Add an expression such as

```
(add-twice 3 5)
```

Click the "Run" button and verify that DrRacket prints the correct answer in the interactions pane (13 for the expression give above).

Create and try out at least two other expressions that use `add-twice`.

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 `(g 3 5)` would be `(+ 3 5)`.

All instances of a parameter in the body are replaced in a single step:

```
(define (h x y) (+ x x x y))
```

The substitution for `(h 10 9)` would be `(+ 10 10 10 9)`.

Ex. 7

Given these definitions:

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

What is the value of this expression? `(* (foo 0) (bar 5 (/ 8 (foo 0))))`

Try to figure it out by hand, then compare to the result calculated by DrRacket.

As we have been doing, when faced with choices of substitutions:

1. apply functions only when all arguments are simple values
2. 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 (sqr 3)) ⇒
(g 4 9) ⇒
(+ 4 9) ⇒
13
```

$g(g(1,3), f(3))$
$= g(1+3, f(3))$
$= g(4, f(3))$
$= g(4, 3^2)$
$= g(4, 9)$
$= 4 + 9$
$= 13$

The general substitution rule is:

`(f v1 ... vn)` ⇒ `exp'`

where `(define (f x1 ... xn) exp)` occurs to the left, and `exp'` is obtained by substituting into the expression `exp`, with all occurrences of the formal parameter `xi` replaced by the value `vi` (for `i` from 1 to n).

Note we are using a pattern ellipsis in the rules for both built-in and user-defined functions to indicate several arguments.

`(f v1 ... vn)` ⇒ `exp'`

where `(define (f x1 ... xn) exp)` occurs to the left, and `exp'` is obtained by substituting into the expression `exp`, with all occurrences of the formal parameter `xi` replaced by the value `vi` (for `i` from 1 to n).

```
(define (foo x y) (* x y (sqr y)))

(foo (- 3 1) (+ 1 2)) ⇒
(foo 2 (+ 1 2)) ⇒
(foo 2 3) ⇒
(* 2 3 (sqr 3)) ⇒
(* 2 3 9) ⇒
54
```

Identifiers

Functions and parameters are named by identifiers, like `f`, `x-ray`, `wHaTeVeR`.

- Identifiers can contain letters, numbers, `-`, `_`, `.`, `?`, `=`, and some other characters.
- Identifiers cannot contain space, brackets of any kind, or quotation marks like `` ` ' " ``.
- Identifiers must contain at least one non-number.

Identifier should be meaningful, where possible. See the style guide.

Observations

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. The following two functions are **not** the same:

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

Ex. 8

Given the definitions, try to determine the value of each expression.
Check your understanding by comparing to what DrRacket gives.

1
```
(define x 4)
(define (f x) (* x x))
(f 3) ⇒ ?
```

2
```
(define (huh? huh?) (+ huh? 2))
(huh? 7) ⇒ ?
```

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

Defining constants

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

```
(define k 3)
(define p (sqr k))
```

The effect of `(define k 3)` is to bind the name `k` to the value `3`.

`(define p (sqr k))` is evaluated in two substitution steps. First, the expression `(sqr k)` is evaluated to give `9`. Second, `p` is bound to that value.

> Advantages of constants

Constants:

- can give meaningful names to useful values (e.g. `interest-rate`, `passing-grade`, and `starting-salary`).
- reduce typing and errors when such values need to be changed.
- make programs easier to understand.

Notes:

- `pi` and `e` are built-in constants.
- Constants can be used in any expression, including the body of function definitions
- Constants are sometimes (incorrectly) called variables. Constants don't change (while the program is running); variables can change. Variables are not used in CS135.

Ex. 9

Given the definitions, try to determine the value of each expression. Check your understanding by comparing to what DrRacket gives.

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

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

Ex. 10

Try out the following lines of code in the definitions pane. If you change the order of the first two lines, what happens and why?

```
(define x (+ 2 3))
(define y (+ x 4.5))
x
y
```

Rule 3: Constants

A constant definition binds a name (the constant) to a value (the value of the expression).

We add the substitution rule:

`id` ⇒ `val`

where `(define id val)` occurs to the left.

To avoid a lot of repetition, we adopt the convention that we stop repeating a definition once its expression has been reduced to a value (since it cannot change after that).

```
(define x 3)
(define y (+ x 1))
y ⇒
(define x 3)
(define y (+ 3 1))
y ⇒
(define x 3)
(define y 4)
y ⇒
(define x 3)
(define y 4)
4
```

```
(define x 3)
(define y (+ x 1))
y ⇒
(define y (+ 3 1))
y ⇒
(define y 4)
y ⇒
4
```

These two examples are the same except that the one on the left does not follow this convention.

Comments

Comments let us write notes to ourselves or other programmers.

Comments start with a semi-colon, `;`, and extend to the end of the line.

```
;; By convention, please use two semicolons, like
;; this, for comments which use a whole line.

(+ 6 7) ; comments after code use one semicolon.

;; Let's define some constants:
(define year-days 365) ; not a leap year
```

Block Comments

Sometimes it's useful to "comment out" a section of a program. There are two options to do this quickly:

- Select the text and use DrRacket's *Racket* → *Comment Out with Semicolons* command
- Use a multi-line comment:

```
#|
(define (function-to-temporarily-remove x y)
    (+ x y))
|#
```

! In DrRacket there is a command *Racket* → *Comment Out with a Box*.
Never use this command! It makes your assignment impossible to mark.

Consider a function to determine the distance from your current location, (c_x, c_y), to the closest of two other locations, (a_x, a_y) or (b_x, b_y):

```
;; Find the distance from (cx,cy) to the closer of two locations,
;; (ax,ay) and (bx,by).
(define (distance-to-closer cx cy ax ay bx by)
    (min (sqrt (+ (sqr (- ax cx)) (sqr (- ay cy))))
         (sqrt (+ (sqr (- bx cx)) (sqr (- by cy))))))

(distance-to-closer 0 0 3 4 5 6)
```

Notice the two instances of nearly identical code.

A better solution is to create a **helper function**, a function that helps implement another function. In this case, the helper function is named `distance`:

```
;; Find the distance from (cx,cy) to the closer of two locations,
;; (ax,ay) and (bx,by).
(define (distance-to-closer cx cy ax ay bx by)
    (min (distance cx cy ax ay)
         (distance cx cy bx by)))

(define (distance x1 y1 x2 y2)
    (sqrt (+ (sqr (- x2 x1)) (sqr (- y2 y1)))))

(distance-to-closer 0 0 3 4 5 6)
```

Helper functions are used for three purposes:

- Reduce repeated code by generalizing similar expressions.
- Factor out complex calculations.
- Give meaningful names to operations.

There are a number of benefits to using helper functions:

- They often (but not always) reduce typing.
- Well-chosen names make programs easier to understand.
- Improvements to the code (bug fix, better performance, better understandability) only need to be applied once.

Helper functions are placed after the function that uses them, although there are exceptions:

- Helpers used to define constants must be defined before being used. For `(define c (distance 1 1 3 9))`, `distance` must already be defined.
- Helpers used in several functions in the same file are often placed first.
- The order of functions specified in an assignment takes precedence over the rules above: functions completed for part (a) will be placed before functions for part (b).

`check-expect` is a special form that we use to test our functions.

```
(check-expect (distance 0 0 3 4) 5)
(check-expect (distance 3 4 0 0) 5)
(check-expect (distance 1 1 4 5) 5)
```

`(check-expect expr-test expr-expected)` consumes two expressions:

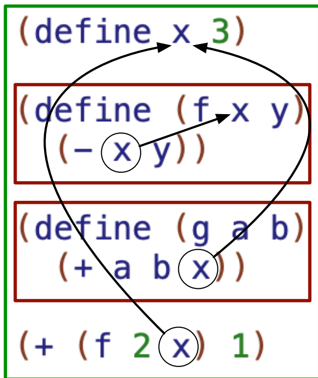
- `expr-test` is the expression (usually a function application) we are testing.
- `expr-expected` is the expected result; the “correct answer”.

So here we are saying that if the `distance` function is properly written, it should be that `(distance 0 0 3 4)` produces `5` and that `(distance 1 1 4 5)` also produces `5`.

This both helps us understand the function, and demonstrate that our code works properly.

We’ll have much more to say about `check-expect` in upcoming modules.

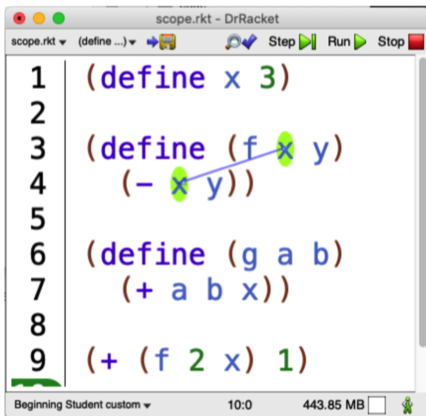
The **scope** of an identifier is where it has effect within the program.

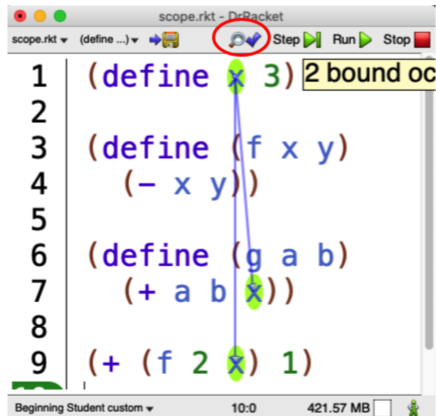


- Two kinds of scope (for now): global and function
- The smallest enclosing scope has priority
- Duplicate identifiers within the same scope will cause an error

```
(define f 3)
(define (f x) (sqr x))
Racket Error: f: this name
was defined...
```

DrRacket can help you identify an identifier's scope.





Use the definitions window:

- Can save and restore your work to/from a file
- Can accumulate 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

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.

- 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.
- You should be able to apply our first three substitution rules to simplify a program to a value.

Ex. 11

Write a Racket function corresponding to

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

(`(sqrt n)` computes $\sqrt{n}$ and `(sqr n)` computes n^2.)

Ex. 12

Evaluate the following program manually to determine what the result should be. Then run it in Racket to check your work:

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) ; ⇒ ?
```

The following functions and special forms have been introduced in this module:

`* + - / abs ceiling check-expect` **`define`** `exp expt floor log max min modulo quotient remainder round sgn sqr sqrt`

You should complete all exercises and assignments using only these and the functions and special forms introduced in earlier modules. The complete list is:

`* + - / abs ceiling check-expect` **`define`** `exp expt floor log max min modulo quotient remainder round sgn sqr sqrt`