Today’s topics
Racket review
A little more arithmetic
The “string” data type
Tokenizing a program
Describing “legal” programs
Announcements

• Register via Banner by 1PM today
• You’ll get email from the TAs about lab signups...
• And soon after, you’ll get email about lab-time assignments
• Homework will come out later today
  • Collaboration policy
  • Some BNF things
  • A little bit about Racket arithmetic expressions
• Challenge problem
Racket Review

• Programs consist of a sequence of zero or more definitions, followed by a zero or one expressions
Examples of programs so far

17
-------
(define height 36)
(define width 11)
height
-----
(+ 3 5)
---
Let’s do more arithmetic as a warmup.
Undefined so far

• Definition
• Expression
  • Number expression
• Reading
• Processing
• Evaluation
• Printing
  • Printed representation
• Value
  • Number value
Examples of programs so far

17
- - - - - - - -
(define height 36)
(define width 11)
height
- - - -
(+ 3 5)
- -
???
Examples of programs so far

17

(define height 36)(

All these labels can be determined by just looking at the text of the program itself!
Which program texts are legal?

• Build up bit by bit
• Use a special notation *unrelated to Racket* to describe legal text and to name their parts
• We’ll be informal about lowest-level details
Structure of a definition (for now)

• Example: `(define height 37)`

• structure: `(define <name> <num>)`

• You don’t yet know the rules for “num”s, but any list of digits works.

• The green things in pointy brackets are descriptions of what has to go in those places, *not* what you actually type.

• The parentheses and the keyword `define` are “literals” – things you must type (almost) exactly as shown
  • You can add blanks before/after parens...but please don’t
  • (Approximate) You *must* have a blank (or other “whitespace”) after `define`
How did I know I could write those things?

• What are the rules of “sentence construction” in this new language?
• How do we write down such rules?
• How do we know what the “sentences” mean when we write them?
• What do “errors” mean/do?
• Let’s address the first two
A first step: breaking a program into "tokens"

- Tokens in a programming language are like words or punctuation marks in a human language: they're the building blocks.
- (CS17) Racket tokens: a few punctuation marks, a few "keywords", numbers, names.
- Other Racket tokens: more punctuation marks, other keywords.
  - Read about them in the Racket documentation if you want to become an expert.
  - Absolutely not necessary for CS17.
  - I have no clear idea what "," or "\" or even "#" mean in Racket, for instance.
  - CS17 is not a Racket course, so this is OK.
Numbers

• Many ways to write these in Racket
  • 526
  • 526.04
  • 2.3e5
  • 2.3e-2
• Too many more to worry about
• We’ll only be using the ones above
A first step: breaking a program into "tokens"

- Tokens in a programming language are like words or punctuation marks in a human language: they're the building blocks.
- (CS17) Racket tokens: a few punctuation marks, a few "keywords", numbers, names, strings,
- The punctuation marks we care about are (, ), [, ], ; ,”
- Others you should avoid (because they may mean things in other parts of Racket that we won’t touch): ` # ' | { }
Keywords

• So far, only define
• Soon to come: and, or, cond, if, let, let*, letrec, lambda, empty, true, false
• Others, too, but none you’re likely to encounter by mistake
Strings

• Text between matching upright double-quotes:
  • "my name is Spike"
  • "cat"
  • ""
    • the “empty string”
  • "17"
    • a string containing the characters “1” and “7”; unrelated to the number 17.

• Never use “smart quotes” – Racket will not understand
• Pro tip: if you only work in Dr Racket, you'll never accidentally produce those!
(let
  ([alon1 (list 1 2)]
   [alon2 (map (lambda (x) (/ x 4.0)) (list 2 14))])
  (map + alon1 alon2))

• Assumption: we can break this into “tokens”: little pieces that constitute the “words” of our language
  • (  
  • let  
  • (  
  • [  
  •   alon1  
  •   (  
  •   list  
  •   1  
  •   2  
  • )  
  • ]  
  • ...
Activity:

• Identify the tokens in these Racket programs
• For each one, say what type it is
• 22e4
• (+ 3 7.2)
• (name1 name2 name3)
Summary (for CS17)

• Special characters: ( ) [ ] ; "
• Keywords: define, ...
• Numbers (things that look like numbers)
• Names (other, non-keyword, non-number stuff, with no whitespace or special characters)
• Strings (stuff between double-quotes)
• Some CS17-specific rules:
  • For multi-word names, separate words with hyphens
  • Use names that help your reader know what you're talking about
  • mostly stick to lower case
From tokens to language

• To “define” a programming language, specify
  • Which things are tokens
  • Which token-sequences are allowed

• The rules of what’s allowed are called “syntax”
  • I’ll gradually disclose these over the next few lectures

• We’ll also assign meaning to each allowable token-sequence; that’s called “semantics”

• For now, the allowable token-sequences (“programs”) are pretty limited: individual numbers, things like (+ 3 5)
Structure of a definition (for now)

• Example: (define height 37)

• structure: (define <name> <num>)
A structured way to write that description

\[
<\text{definition}> := ( \text{define} \ <\text{name}> \ <\text{num}> )
\]

- **Underlined** stuff is literal: you have to have exactly those characters

- For Racket, I’m going to be sloppy about whitespace: you need it wherever it helps separate things, but can skip it where separation is obvious (e.g., a left-paren *is* a single token, whether there’s white space around it or not)
  - There’s a big difference between 2 2 (two number tokens) and 22 (one number token)
A structured way to write that description

<definition> := ( define <name> <num> )

• Underlined stuff is literal: you have to have exactly those characters

• Things in pointy-brackets are names for parts of your program.

• We read this aloud as “a definition consists of a left-paren, the keyword “define”, a name, a number, and a right-paren”

• I can now write down several things and ask whether you think that they are “definitions” according to this rule
Are these definitions?

• Using this rule:
  • \(<\text{definition}\> := (\ \text{define} \ \text{name} \ \text{num}\>)\)
  • (\text{define} \ \text{height} \ 37)\)
  • (\text{define} \ \text{height})\)
  • (\text{define} \ \text{height} \ \text{width})\)
  • (\text{def} \ h -7)\)
  • \text{define} \ \text{height} \ 11
A small fix

• The rule I just gave for definitions wasn’t the real one, but it was nice and compact and self-contained.

• The real rule is

<definition> := ( define <name> <expression> )

• Less satisfying, because we don’t yet know what <expression> actually means.
Simplifying

- To keep the stuff I’m writing all on one slide, I’ll abbreviate

\[
\text{<defn>} := (\text{define} \ \text{<name>} \ \text{<expr>})
\]
Added features of our notation

• A “star” after something means “any number of these, including zero”
• So e* would mean “zero or more copies of the letter e”
• Square brackets around something mean “optional” (either zero or one of these).
• So k[e]en would means “either ken or keen”
From last time

• Programs consist of a sequence of zero or more *definitions*, followed by a zero or one *expressions*

• In this new notation, we could write

\[ \langle \text{prog} \rangle := \langle \text{defn} \rangle^* [\langle \text{expr} \rangle] \]

• Now all we need to do is say what “expr” means, and we’ll be done
BNF

• This green pointy-bracket notation is call “Backus-Naur Form,”
• It’s used all over computer science to describe things that have pattern-oriented structures
• Analogous stuff has been used to describe the branching structure of (physical) trees!