EECS 111

Programming with **text**

Textual languages

- Diagrams are good for making the **flow of data** explicit in programs
  - But they can be **cumbersome** for large programs
  - And there are some techniques they **don't express well**
- So the vast majority of programming languages are **textual**
- We'll be teaching you the basics of Racket today

Names

- Any language (Racket, C++, English) has to provide some mechanism for **naming things**

- In programming languages, those things are generally **data objects**
  - Numbers
  - Procedures
  - Strings
  - Etc.

Constants

- Names whose **spelling** determines the object being referred to
  - Programmer can't change their meaning
- The most basic kind of name
- **Numbers**
  - Sequence of digits means a number
  - Can also include decimal point and/or sign
  - 1, 7, -2.5
- **Strings**
  - Any text enclosed in " " names a text string
  - "this is a string"
- We'll see **other kinds** of constants later

Variables

- **Arbitrary** names that the programmer can use to denote anything they want
- Can refer to **different objects** at different times
- Many variables come **predefined**:
  - string-append, +, -, *

- Any sequence of **letters**, **numbers**, and **most punctuation marks**, that **doesn't look like a number**
  - **A**, **b**, **c**, **test**, **bla**, **foo**, **x**, **x1**
  - **+**, **-**, *****, **/**
  - **this-is-a-variable-name**, **As-is-this**
- **Case-sensitive**: x is different from X
- **+1** is a **number**, **1+** is a **variable name**

Defining new variables

(define *name value*)

- Tells system that *name* now refers to *value*
  - *Name* must be a valid variable name
  - But *value* can be an arbitrary expression
- Has to be executed to take effect
- **Naming** is the most basic **abstraction** mechanism

Procedure calls

- To call a procedure, write the **procedure**, followed by its **inputs**, separates by **spaces**:
  - ***procedure input$_1$ … input$_n$***
- Then **wrap it in parentheses**:
  (***procedure input$_1$ … input$_n$***)
- **Line breaks** and other **extra whitespace** are fine

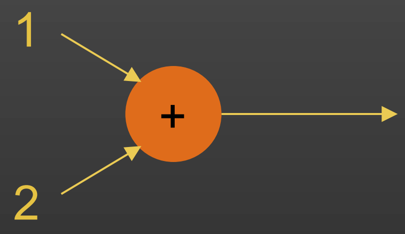
Examples

- (+ 1 2)
- (+ 1 2 3)
- (string-append " this is "
  "a test")
- (string-append "this is " "a test")
- (string-append "a" "b" "c")

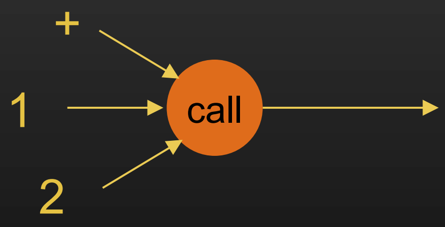
Note: + and string-append can allow variable numbers of inputs in Racket

Equivalent diagrams

(+ 1 2) means:



or really:



Nested (chained) calls

- The **basic call** format is:
  - (*procedure input*$_1$ … *input*$_n$)
- But **any** of these **can be calls** themselves
- This means the **output** of the **inner call** is used as the **input** to the **enclosing call**
- Calls are **chained** by **nesting** their expressions

Examples

-
  ```
  (+ 1 (+ 1 2))
  ```

-
  ```
  (string-append
     "this "
     (string-append "is a "
                    "test"))
  ```

-
  ```
  (square 10 "solid" "blue")
  ```

-
  ```
  (above (square 10
                 "solid" "blue")
         (square 10
                 "solid" "red"))
  ```

Parentheses always mean call (for now)

There's always exactly **one call for every pair of parens**

(+ 1 2) means:

$$\begin{tikzcd}
+ \arrow[dr] & & \\
1 \arrow[r] & \text{call} \arrow[r] & {} \\
2 \arrow[ur] & &
\end{tikzcd}$$

(+ (1) 2) means:

$$\begin{tikzcd}
 & + \arrow[dr] & & \\
1 \arrow[r] & \text{call} \arrow[r] & \text{call} \arrow[r] & {} \\
 & 2 \arrow[ur] & &
\end{tikzcd}$$

((+ 1 2)) means:

$$\begin{tikzcd}
+ \arrow[dr] & & & \\
1 \arrow[r] & \text{call} \arrow[r] & \text{call} \arrow[r] & {} \\
2 \arrow[ur] & & &
\end{tikzcd}$$

Procedure position vs argument position

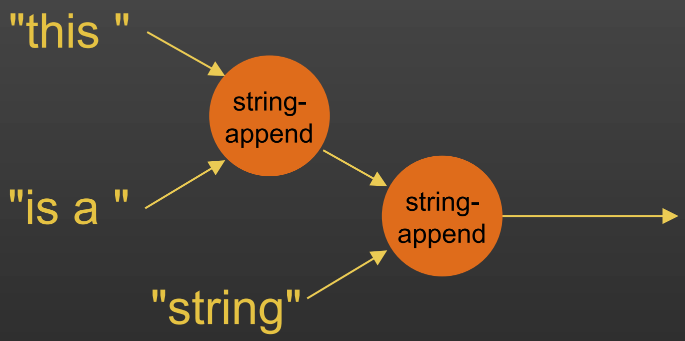
- The **procedure** to call is always the **first item** after the parens
- The **rest** of the items are always **inputs**

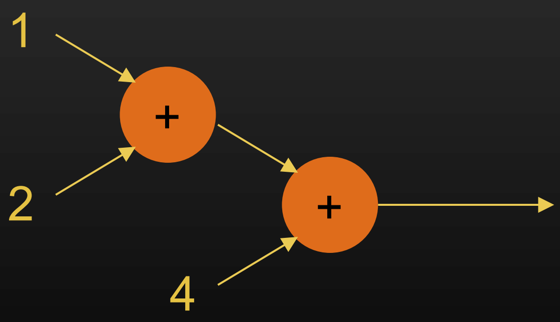
- Placing something that's **not a procedure at the beginning** causes a **not a procedure exception**

- (+ 1 2)
    - Call + with 1 and 2 as inputs
- **(1 + 2)**
    - Call 1 with + and 2 as inputs
    - Not a procedure exception

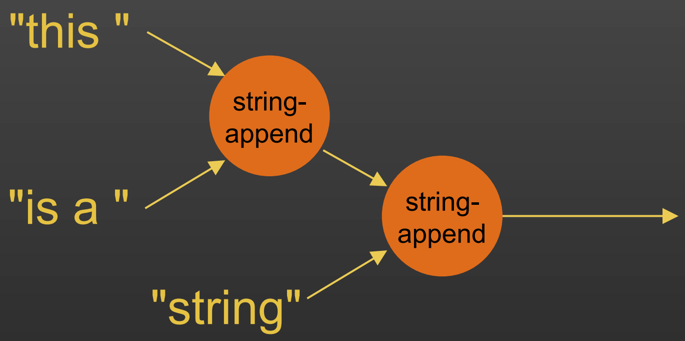
- (+ (+ 1 2) 3)
    - Call + with 1 and 2 as inputs
    - Call + again with previous result and 3 as inputs
- **((+ 1 2) 2)**
    - Call 3 (the output of (+ 1 2)) with 2 as an input
    - Not a procedure exception

What are the textual versions of these DFDs?

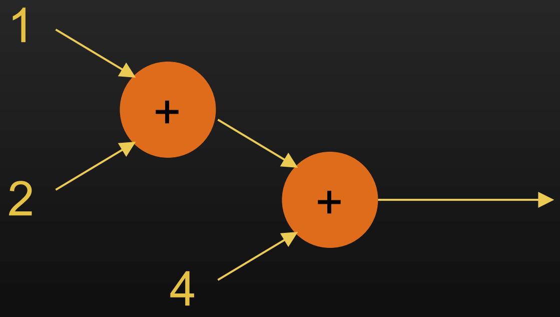




What are the textual versions of these DFDs?

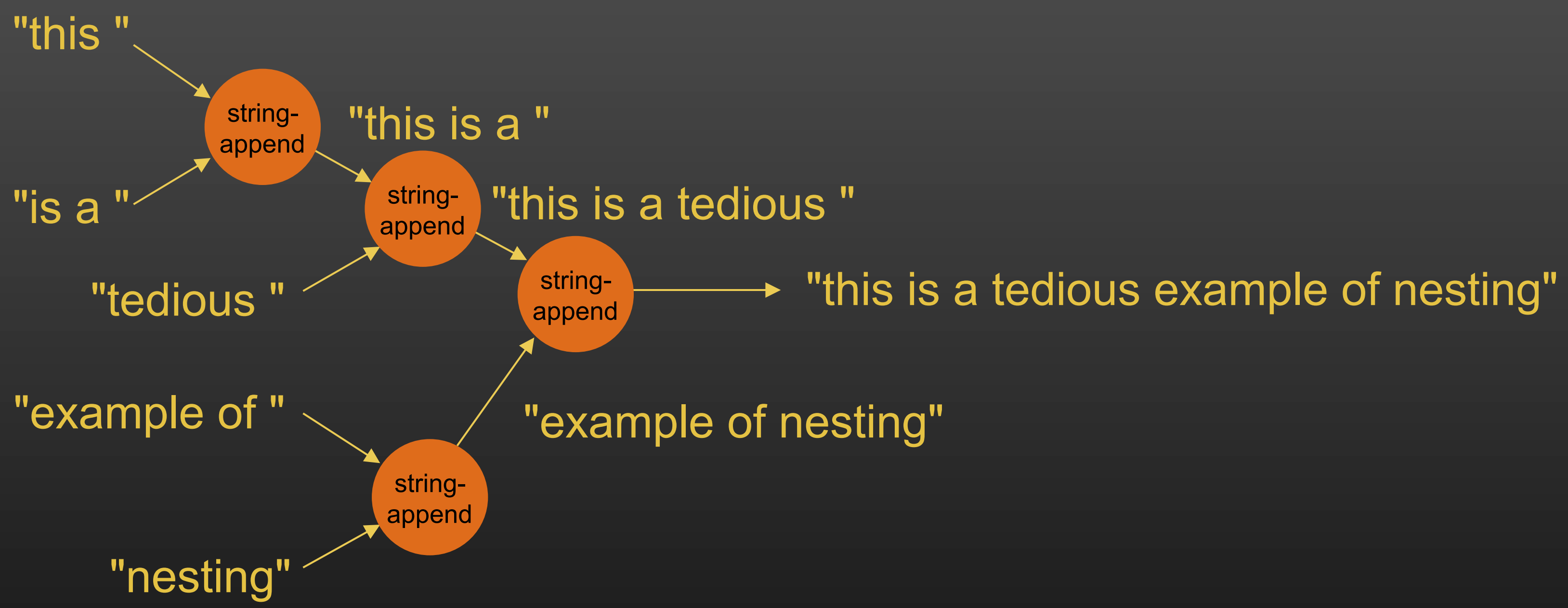


```
(string-append (string-append
  "this " "is a ") "string")
```

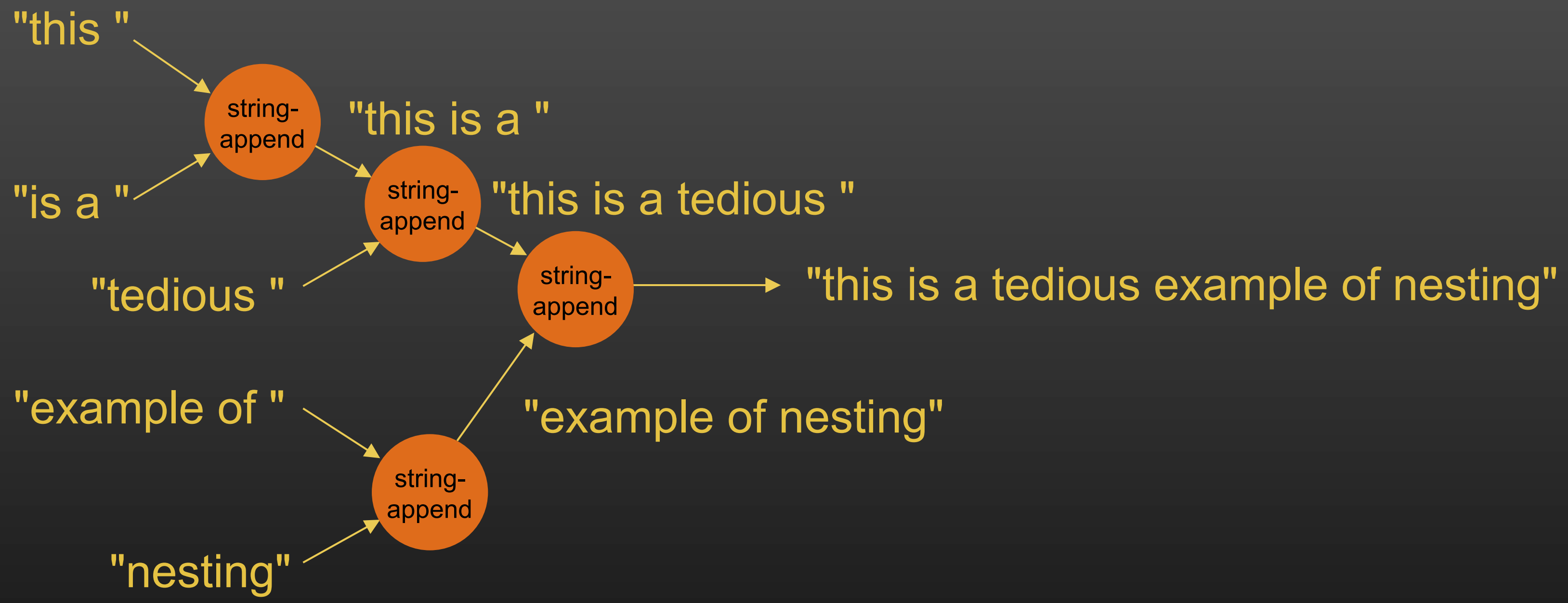


```
(+ (+ 1 2) 4)
```

What’s the expression for this DFD?



What’s the expression for this DFD?



(string-append (string-append (string-append "this " "is a ") "tedious ") (string-append "example of " "nesting"))

Making it legible

```
(string-append (string-append (string-append "this " "is a ")
   "tedious ") (string-append "example of " "nesting"))
```

- This is completely **illegible**

Making it legible

```
(string-append (string-append (string-append "this "
                                             "is a ")
                              "tedious ")
               (string-append "example of "
                              "nesting"))
```

- To **make it legible**, we:
  - Break it into **multiple lines**
  - **Indent** it to **align inputs** to the same call
- In other words, we put it in **outline form**

Making it mutually intelligible

- The **computer** ignores
  - Line breaks
  - Extra whitespace
  - It only **looks at the parens**
- **Humans** ignore
  - The parens (or at least we stink at reading them)
  - We **look at the indentation**
- Note: this means it's **critical** the **indentation match the parentheses**!

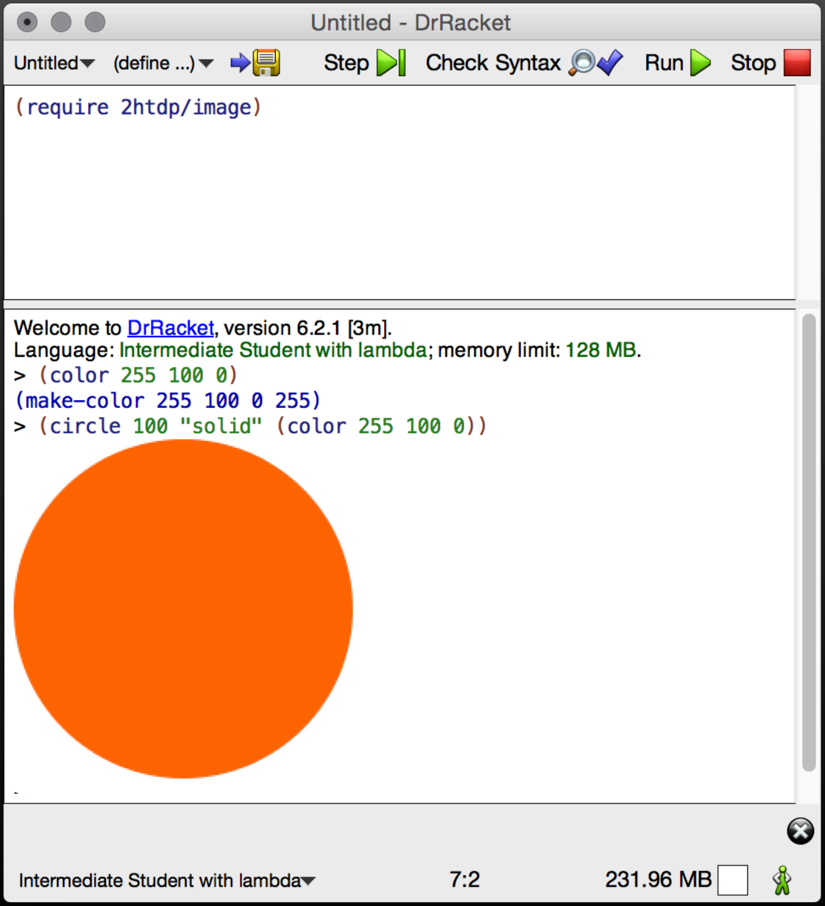
```
(string-append (string-append (string-append "this "
                                                "is a ")
                                 "tedious ")
               (string-append "example of "
                                 "nesting"))
```

Keeping your code indented

- Racket will **automatically indent** lines
  - When you hit **return/enter** (indents the **new line**)
  - When you hit **tab** (reindents **current line**)
- If you run into a problem, one of the **first things to do** is to have racket **reindent** your code
  - Make sure that you and racket have the same idea of what is an input to what

Using DrRacket

- The **top (or left)** pane is a **file window (also called the 'definitions pane')**
  - Code here doesn't run until you choose **Run** from the **Racket** menu
- The **bottom** pane is an interaction window
  - Aka a **REPL** (Read/ Evaluate/Print Loop)
  - You can type expressions here at the **">" prompt**
  - Racket will **run** them and **print the result**



```
(require 2htdp/image)
```

```
Welcome to DrRacket, version 6.2.1 [3m].
Language: Intermediate Student with lambda; memory limit: 128 MB.
> (color 255 100 0)
(make-color 255 100 0 255)
> (circle 100 "solid" (color 255 100 0))
```

Remember!

- Don't write you code as **one long line**
  - **Add line breaks** between arguments in complex calls
- Always **keep your code indented** properly
  - Press **tab** to ask Racket to reindent a line based on the surrounding parentheses
  - If things that should be inputs to the same procedure **don't line up**, then your **parentheses are wrong**