Problem Solving and Abstraction (CMPU 101)

Tom Ellman

Lecture 2
Names (a.k.a. Identifiers)

- A sequence of characters:
  - Letters: a ... z or A ... Z
  - Numerals 0 ... 9
  - Punctuation: - (dash) or _ (underscore)

- Starting with a letter.
  
  **OK**: x, y1, CAT, dog, pet-store, dog_food
  
  **Not OK**: 137student, s!, t: , u*, $
Names

```python
>>> x = 17
>>> y = 3
>>> z = x + y
>>> z
20
>>> title = "President"
>>> sir_name = "Bradley"
>>> titled_name = title + " " + sir_name
>>> titled_name
"President Bradley"
```
Definitions

\[ x = 17 \text{ and } \text{title} = \text{“President”} \text{ are definitions.} \]

A definition creates a binding that associates a name with a value by storing them in Pyret’s internal directory.

If Pyret comes across the name later, Pyret will replace the name with its associated value.

In Pyret, definitions are sometimes called declarations. In other languages, e.g., Java, C++ these are different concepts.
Definitions versus Expressions

• Notice that Pyret does not display a value after processing a definition.
• Definitions are statements, not expressions.
• Statements are not evaluated to produce values.
• Instead, statements cause side effects of changing Pyret’s directory.
Binding and Reference

*Bind* values to names:

*Refer* to values using names:

```python
>>> x = 17
>>> y = 3
>>> z = x + y
>>> z
20

>>> title = "President"

>>> sir-name = "Bradley"

>>> titled-name = title + " " + sir-name

>>> titled-name
"President Bradley"
```
Pyret’s Internal Directory

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>17</td>
</tr>
<tr>
<td>y</td>
<td>3</td>
</tr>
<tr>
<td>z</td>
<td>20</td>
</tr>
<tr>
<td>title</td>
<td>“President”</td>
</tr>
<tr>
<td>sir-name</td>
<td>“Bradley”</td>
</tr>
<tr>
<td>titled-name</td>
<td>“President Bradley”</td>
</tr>
</tbody>
</table>

Notice that the directory stores values, not expressions:

- the value 20 rather than the expression $x + y$
- and
- the value “President Bradley” rather than the expression `title + " " + sir-name`. 
Substitution

When Pyret evaluates the expression:

\[ x + y \]

it finds the values of \( x \) an \( y \) in the directory and replaces \( x \) and \( y \) with their respective values to get:

\[ 20 \]
“foo” is a string.

foo (without quotes) is potentially a name.

Since foo has not been defined, Pyret says it is unbound.

An error occurs if we try to use foo as a name before we have given it a definition.
Once we define a name, we (normally) cannot change its value. If we try to do so, we get an *shadow* error.

```python
>>> favorite_color = "red"
>>> favorite_color
"red"
>>> favorite_color = "blue"

The declaration of `favorite_color` shadows a previous declaration of `favorite-color`
Try These Expressions!

```python
include image
base = rectangle(20, 20, "solid", "blue")
base
roof = triangle(30, "solid", "red")
roof
house = above(roof, base)
house
neighbors = beside(house, house)
neighbors
```
Building Expressions from Sub-Expressions

```python
>>> include image

>>> base = rectangle(20, 20, "solid", "blue")

>>> base

>>> roof = triangle(30, "solid", "red")

>>> roof

>>> house = above(roof, base)

>>> house

>>> neighbors = beside(house, house)

>>> neighbors
```

- Sub-Sub-Expression
- Sub-Sub-Expression
- Sub-Expression
- Expression
Cut and Paste into the Definitions Pane & Press Run

include image

base = rectangle(20, 20, "solid", "blue")
roof = triangle(30, "solid", "red")
house = above(roof, base)
beside(house, house)
Definitions

```plaintext
use context essentials2021
include image
base = rectangle(20, 20, "solid", "blue")
roof = triangle(30, "solid", "red")
house = above(roof, base)
beside(house, house)
```

Interactions
We are typing and evaluating similar expressions over and over. This could get tedious.

In each case, we compute a sum and divide by two.

We can capture this pattern in a function definition.

DRY Principle: Don't Repeat Yourself.
Defining a Function

Function definition creates a new binding in the Pyret’s directory between the function name and its definition. The body is not evaluated at this time.

fun average(x,y): (x + y) / 2 end
Functional Abstraction

We replace the common parts of these expressions by new names, called **formal parameters**.
Function Definition Format

fun <function name>(<parameters>):
    <body expression>
end

fun average(x,y):
    (x + y) / 2
end
```python
>>> fun average(x, y): (x + y) / 2 end

>>> average(45, 63)
54

>>> average(17, 137)
77

>>> average(123, 321)
222
```
Pyret Evaluating: average(45,63)

1. Find the definition of average in the directory.
2. Associate the formal parameters \((x,y)\) in the definition with the actual parameters \((45,63)\) and create temporary bindings: \(x = 45\) and \(y = 63\).
3. Evaluate the body expression \(\frac{x + y}{2}\) using the new bindings for \(x\) and \(y\) to get the value 54 of average(45,63).
Repeated Similar Expressions

We are typing and evaluating similar expressions over and over. This could get tedious.

How are the expressions similar? How are they different?

Can can capture this pattern in a function definition?

DRY Principle: Don't Repeat Yourself.
fun house(base-color,roof-color):
    above(triangle(30, "solid", roof-color),
    rectangle(20,20,"solid",base-color))
end

>>> house("blue","red")

>>> house("black","orange")

>>> house("violet","yellow")
Annotations and Contracts

```rust
fun house(base-color :: String,
          roof-color :: String) -> Image:
    above(triangle(30,"solid",roof-color),
           rectangle(20,20,"solid",base-color))
end
```

We use :: String to require the user of house to provide only data of type String for base-color and roof-color.

We use -> Image to promise to the user that the house function will return a datum of type Image.

The requirement on the user of house and the promise made by the programmer are a contract. If both full the terms of the contract there will not be any data-type errors.
Pyret detects the data-type error while evaluating the expression: `house("blue",7)`.
Solving the Quadratic Equation

\[ a x^2 + b x + z = 0 \]

Solve for \( x \)

quad-root-high \quad \text{High Root}
quad-root-low \quad \text{Low Root}
fun quad-high(a :: Number,
    b :: Number,
    c :: Number) -> Number:
    ((0 - b) + num-sqrt((b * b) - (4 * (a * c)))) / (2 * a)
end
fun quad-high(a :: Number,  
    b :: Number,  
    c :: Number) -> Number:  
    ((0 - b) + num-sqrt((b * b) - (4 * (a * c)))) / (2 * a)  
where:  
    quad-high(1,0,-4) is 2  
    quad-high(1,-5,4) is 4  
end
fun quad-low(a :: Number,
    b :: Number,
    c :: Number) -> Number:
    ((0 - b) - num-sqrt((b * b) - (4 * (a * c)))) / (2 * a)
end
fun quad-low(a :: Number,
    b :: Number,
    c :: Number) -> Number:
    ((0 - b) - num-sqrt((b * b) - (4 * (a * c)))) / (2 * a)
where:
    quad-low(1,0,-4) is -2
    quad-low(1,-5,4) is 1
end