Evaluating Functions and Conditionals

24 January 2024
Assignment 1 will come out tomorrow (by 5 p.m.) and be due on Wednesday (by 11:59 p.m.).
Computer Science I: Problem-Solving and Abstraction

CMPU-101 §52
Vassar College, Spring 2024

Navigation  Home  Labs  Assignments  Resources  Gradescope

Coaching

I will make every effort to give each of you the attention and feedback you need to be successful in this course – but there’s only one of me! Therefore, I rely on the coaches to help me help answer your questions.

In addition to working during our labs each week, each coach will be available to help you in the Agile Lab (SC 006) at scheduled times.

Important: The coaches are prohibited from giving you the solutions to labs and assignments, but they are able to guide you as you work to solve your programming tasks. When this works well, they will help you answer your own questions!

Coaching Calendar
Where are we?
We’ve been using Pyret to write expressions using data, including

- **numbers** like 0, -10, and 0.4;
- **strings** like "", "hi", and "111"; and
- **images** like ●, a.k.a., `circle(2, "solid", "red")`,

which we modify or combine using operators like + and * and functions like `string-append` and `above`. 
f(42)
What function to call

\[ f(42) \]
What function to call

Argument to the function

$f(42)$
“Call f on 42.”
What function to call

First argument

Second argument

num-max(13, 42)
Distinguishing types of data helps to catch mistakes.

If you try to give

   a string to / or

   a number to overlay,

we want Pyret to catch the problem right away, giving a helpful error message.
Fail-fast system

In systems design, a fail-fast system is one which immediately reports at its interface any condition that is likely to indicate a failure. Fail-fast systems are usually designed to stop normal operation rather than attempt to continue a possibly flawed process. Such designs often check the system’s state at several points in an operation, so any failures can be detected early. The responsibility of a fail-fast module is detecting errors, then letting the next-highest level of the system handle them.

Hardware and software

Fail-fast systems or modules are desirable in several circumstances:

- Fail-fast architectures are based on an error handling policy where any detected error or non-considered state makes the system fail (fast). In some sense the error handling policy is the opposite of that used in a fault-tolerant system. In a fault-tolerant system an error handling policy is established to have redundant components and move computation requests to alive components when some component fails. Paradoxically fail-fast systems make fault-tolerant systems more resilient. We can have 10 redundant servers for a given database, but if the shared configuration for the 10 servers is updated with wrong authentication data for clients, all of them will "redundantly fail". In that sense, a fail-fast system will get sure that all the 10 redundant servers fail as soon as possible to make the DevOps react fast.
- Fail-fast components are often used in situations where failure in one component might not be visible until it leads to failure in another component as a consequence of lazy initialization. e.g. "The system that is "doomed" to fail because a file-system path is wrongly setup, does it fail at startup because the file-system path is not checked at startup. Only when a client-request arrives the system fails, at random, later on.
- Finding the cause of a failure is easier in a fail-fast system, because the system reports the failure with as much information as possible as close to the time of failure as possible. In a fault-tolerant system, the failure might go undetected, whereas in a system that is neither fault-tolerant nor fail-fast the failure might be temporarily hidden until it causes some seemingly unrelated problem later.
- A fail-fast system that is designed to halt as well as report the error on failure is less likely to erroneously perform an irreversible or costly operation.

Developers also refer to code as fail-fast if it tries to fail as soon as possible at variable or object initialization. In object-oriented
We’ve seen that we can create more complicated programs by composing function calls, e.g.,

\[ 1 + (2 \div 3) \]

or

```
string-append("hello ",
   string-append("Pyret ", "world!"))
```
And we can give a name to the result of an expression, e.g.,

\[ total = 2 + 3 \]
And we can give a name to the result of an expression, e.g.,

\[ total = 2 + 3 \]

<table>
<thead>
<tr>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>total</td>
</tr>
</tbody>
</table>
And we can give a name to the result of an expression, e.g.,

\[
\text{total} = 2 + 3 \\
\rightarrow \text{total} = 5
\]

Directory

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td></td>
</tr>
</tbody>
</table>
And we can give a name to the result of an expression, e.g.,

\[
\text{total} = 2 + 3 \quad \rightarrow \quad \text{total} = 5
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>5</td>
</tr>
</tbody>
</table>
And we can give a name to the result of an expression, e.g.,

\[
total = 2 + 3 \\
\rightarrow total = 5
\]

\[
new-total = total + 1
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>5</td>
</tr>
</tbody>
</table>
And we can give a name to the result of an expression, e.g.,

\[
\begin{align*}
total & = 2 + 3 \\
\rightarrow total & = 5 \\
new-total & = total + 1
\end{align*}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>5</td>
</tr>
<tr>
<td>new-total</td>
<td></td>
</tr>
</tbody>
</table>
And we can give a name to the result of an expression, e.g.,

\[
\begin{align*}
  total &= 2 + 3 \\
  \rightarrow total &= 5 \\
  new-total &= total + 1
\end{align*}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>5</td>
</tr>
<tr>
<td>new-total</td>
<td></td>
</tr>
</tbody>
</table>
And we can give a name to the result of an expression, e.g.,

\[
\text{total} = 2 + 3 \\
\rightarrow \text{total} = 5
\]

\[
\text{new-total} = \text{total} + 1 \\
\rightarrow \text{new-total} = 5 + 1
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>5</td>
</tr>
<tr>
<td>new-total</td>
<td></td>
</tr>
</tbody>
</table>
And we can give a name to the result of an expression, e.g.,

\[
\begin{align*}
  \text{total} &= 2 + 3 \\
  \quad &\Rightarrow \text{total} = 5 \\

  \text{new-total} &= \text{total} + 1 \\
  \quad &\Rightarrow \text{new-total} = 5 + 1 \\
  \quad &\Rightarrow \text{new-total} = 6
\end{align*}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>5</td>
</tr>
<tr>
<td>new-total</td>
<td></td>
</tr>
</tbody>
</table>
And we can give a name to the result of an expression, e.g.,

\[
\begin{align*}
total &= 2 + 3 \\
       &\rightarrow total = 5 \\

new-total &= total + 1 \\
           &\rightarrow new-total = 5 + 1 \\
           &\rightarrow new-total = 6
\end{align*}
\]
Defining and evaluating functions
Remember functions from middle-school math:

Given $f(x) = \cos(x) + 2$

$f(0) = 1 + 2 = 3$

*The parameter $x$ stands for varying values*
Pyret functions work much the same way:

```pyret
fun f(x): num-cos(x) + 2 end
```

```
f(0)
→ num-cos(x) + 2
→ num-cos(0) + 2
→ 1 + 2
→ 3
```

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Note that the parameter names are only defined inside the function body:

```plaintext
>>> fun f(x): num-cos(x) + 2 end
>>> f(0)
3
>>> x
Error!
```

Once the function is finished, the names are removed from the directory.
We say a parameter name has only *local scope*, while names defined outside a function have *global scope*. 
Example
Mary Berry needs to know how many cakes to bake for her cake shop.

To avoid running out or having too many, she wants to bake two cakes more than the number she sold the previous day.

E.g., if Mary sells eight cakes on Monday, she makes ten cakes on Tuesday.

Let’s write some code to help Mary.
fun cakes-to-make(num-sold):
    num-sold + 2
end
Keyword to define a function

fun cakes-to-make(num-sold):
    num-sold + 2
end
fun cakes-to-make(num-sold):
    num-sold + 2
end
fun cakes-to-make(num-sold):
    num-sold + 2
end
fun cakes-to-make(num-sold):
    num-sold + 2
end
fun cakes-to-make(num-sold):
    num-sold + 2
end
fun cakes-to-make(num-sold):
    num-sold + 2
end
fun cakes-to-make(num-sold):
    num-sold + 2
end

How to transform the data
fun cakes-to-make(num-sold):
    num-sold + 2
end
fun cakes-to-make(num-sold):
    num-sold + 2

_keyword to signal the end of the function definition_
fun cakes-to-make(num-sold):
    num-sold + 2
end
Functions are abstractions over specific computations
# Draw a traffic light

```plaintext
above(circle(40, "solid", "red"),
    above(circle(40, "solid", "yellow"),
        circle(40, "solid", "green")))
```
# Draw a traffic light

above( circle(40, "solid", "red"),
above(circle(40, "solid", "yellow"),
circle(40, "solid", "green"))

![Traffic Light](image-url)
# Draw a traffic light

above( circle(40, "solid", "red"),
      above( circle(40, "solid", "yellow"),
             circle(40, "solid", "green") )
)

Unchanging  Varying
# Draw a traffic light
above(circle(40, "solid", "red"),
    above(circle(40, "solid", "yellow"),
        circle(40, "solid", "green"))))
# Draw a traffic light
above(circle(40, "solid", "red"),
    above(circle(40, "solid", "yellow"),
        circle(40, "solid", "green"))))

# Can be changed to
fun bulb(color):
    circle(40, "solid", color)
end

above(bulb("red"),
    above(bulb("yellow"),
        bulb("green")))
fun bulb(color):
    circle(40, "solid", color)
end

above(bulb("red"),
    above(bulb("yellow"),
        bulb("green")))
fun bulb(color):
    circle(40, "solid", color)
end

fun traffic-light():
    above(bulb("red"),
         above(bulb("yellow"),
                bulb("green")))
end
Remember: Each function has one job!
Example
For Mary’s cake shop, we want to determine the price of each cake based on the cost of the ingredients and the time to prepare it.

As the price, she uses twice the cost of the ingredients plus \( \frac{1}{4} \) of the preparation time in minutes.
For Mary's cake shop, we want to determine the price of each cake based on the cost of the ingredients and the time to prepare it.

As the price, she uses twice the cost of the ingredients plus \( \frac{1}{4} \) of the preparation time in minutes.

**Chocolate cake**

*Ingredients:* $10

*Prep. time:* 20 min.
For Mary's cake shop, we want to determine the price of each cake based on the cost of the ingredients and the time to prepare it.

As the price, she uses twice the cost of the ingredients plus \( \frac{1}{4} \) of the preparation time in minutes.

**Chocolate cake**

*Ingredients:* 
$10

*Prep. time:* 20 min.

\[
\text{choc-cake-price} = (2 \times 10) + \left(\frac{1}{4} \times 20\right)
\]
For Mary's cake shop, we want to determine the price of each cake based on the cost of the ingredients and the time to prepare it.

As the price, she uses twice the cost of the ingredients plus \( \frac{1}{4} \) of the preparation time in minutes.

\[
\text{choc-cake-price} = (2 \times 10) + \left(\frac{1}{4} \times 20\right)
\]

**Chocolate cake**
- Ingredients: $10
- Prep. time: 20 min.

**Cheesecake**
- Ingredients: $15
- Prep. time: 36 min.
For Mary's cake shop, we want to determine the price of each cake based on the cost of the ingredients and the time to prepare it.

As the price, she uses twice the cost of the ingredients plus $1/4$ of the preparation time in minutes.

**Chocolate cake**
- Ingredients: $10$
- Prep. time: 20 min.

\[
\text{choc-cake-price} = (2 \times 10) + \left(\frac{1}{4} \times 20\right)
\]

**Cheesecake**
- Ingredients: $15$
- Prep. time: 36 min.

\[
\text{cheesecake-price} = (2 \times 15) + \left(\frac{1}{4} \times 36\right)
\]
We use functions to avoid repetitive code when we need to perform the same operations on different values.

\[
\text{choc-cake-price} = (2 \times 10) + (1/4 \times 20)
\]

\[
\text{cheesecake-price} = (2 \times 15) + (1/4 \times 36)
\]
We use functions to avoid repetitive code when we need to perform the same operations on different values.

\[
\text{choc-cake-price} = (2 \times 10) + \left(\frac{1}{4} \times 20\right)
\]

\[
\text{cheesecake-price} = (2 \times 15) + \left(\frac{1}{4} \times 36\right)
\]
We use functions to avoid repetitive code when we need to perform the same operations on different values.

\[
(2 \times \text{ingredients-cost}) + (1/4 \times \text{prep-time})
\]

\[
\text{choc-cake-price} = (2 \times 10) + (1/4 \times 20)
\]

\[
\text{cheesecake-price} = (2 \times 15) + (1/4 \times 36)
\]
fun cake-price(ingredients-cost, prep-time):
    (2 * ingredients-cost) + (1/4 * prep-time)
end
fun cake-price(ingredients-cost, prep-time):
    (2 * ingredients-cost) + (1/4 * prep-time)
end

The **parameters** are the values passed into the function that it needs to know for each operation.
fun cake-price(ingredients-cost, prep-time):
    (2 * ingredients-cost) + (1/4 * prep-time)
end

Expression repeated each time the function is called
fun cake-price(ingredients-cost, prep-time):
    (2 * ingredients-cost) + (1/4 * prep-time)
end

# Price of chocolate cake
cake-price(10, 20)

# Price of cheesecake
cake-price(15, 36)

To calculate the price of chocolate cake or cheesecake, you just call your function and pass in the relevant values!
Improving our function definitions
fun cake-price(ingredients-cost :: Number, prep-time :: Number):
  (2 * ingredients-cost) + (1/4 * prep-time)
end

We specify the type of each parameter so that Pyret will check that we pass in the right kind of values, just like for built-in operations like + and above.
fun cake-price(ingredients-cost :: Number, prep-time :: Number) -> Number:
   (2 * ingredients-cost) + (1/4 * prep-time)
end

And we can specify the type of value the function returns.
use context essentials2021

fun cake-price(ingredients-cost :: Number, prep-time :: Number) -> Number:
  (2 * ingredients-cost) + (1/4 * prep-time)
end

>>> cake-price(2, 3)
4.75

>>> cake-price("banana", "bundt")

The Number annotation was not satisfied by the value "banana"
(Show program evaluation trace...)
fun cake-price(ingredients-cost :: Number, prep-time :: Number) -> Number:
    doc: "Calculate price of cake based on ingredient cost and preparation time"
    (2 * ingredients-cost) + (1/4 * prep-time)
end

Additionally, a docstring explains what the function does.
fun cake-price(ingredients-cost :: Number, prep-time :: Number) -> Number:
  doc: "Calculate price of cake based on ingredient cost and preparation time"
  (2 * ingredients-cost) + (1/4 * prep-time)
where:
  # Price of chocolate cake
  cake-price(10, 20) is (2 * 10) + (1/4 * 20)
  # Price of cheesecake
  cake-price(15, 36) is (2 * 15) + (1/4 * 36)
end
use context essentials

fun cake-price(ingredients-cost :: Number, prep-time :: Number) -> Number:
  (2 * ingredients-cost) + (1/4 * prep-time)

where:
  # Price of chocolate cake
cake-price(10, 20) is (2 * 10) + (1/4 * 20)
  # Price of cheesecake
cake-price(15, 36) is (2 * 15) + (1/4 * 36)

end
use context essentials2021

fun cake-price(ingredients-cost :: Number, prep-time :: Number) -> Number:
  (2 * ingredients-cost) + (1/3 * prep-time)
where:
  # Price of chocolate cake
  cake-price(10, 20) is 2 * 10 + (1/4 * 20)
  # Price of cheesecake
  cake-price(15, 36) is 2 * 15 + (1/4 * 36)
end

0 TESTS PASSED  2 TESTS FAILED

cake-price

0 out of 2 tests passed in this block.
10 results for "David Bowie"

David Bowie

- David Bowie - Diamond Dogs
  - Price: $24.99
- David Bowie - Low
  - Price: $39.99
- David Bowie - Station To Station
  - Price: $29.99
10 results for "David Bowie"

David Bowie

Sort by: Relevance

- David Bowie - Diamond Dogs $24.99
- David Bowie - Low $39.99
- David Bowie - Station To Station $29.99
fun rectangle-area(r):
    image-height(r) * image-width(r)
end
fun rectangle-area(r :: Image) -> Number:
    doc: "Return the rectangular area of the image"
    image-height(r) * image-width(r)
where:
    rectangle-area(rectangle(0, 0, "solid", "black"))
    is 0
    rectangle-area(rectangle(2, 3, "outline", "blue"))
    is 6
end
fun rectangle-area(r :: Image) -> Number:
  doc: "Return the rectangular area of the image"
  image-height(r) * image-width(r)
where:
  tiny = rectangle(0, 0, "solid", "black")
  rectangle-area(tiny) is 0

  blue = rectangle(2, 3, "outline", "blue")
  rectangle-area(blue) is 6
end
Booleans and \textit{if} expressions
true
false
We can compare values using these operators

<  less than
<=  less than or equal to
>  greater than
>=  greater than or equal to
==  equal to
<>  not equal to

which produce true or false as a result.
Be careful:

\[ x = 2 \]

is assigning the name \( x \) to have the value 2 in the directory.

\[ x == 2 \]

is asking the question “is \( x \) equal to 2?”
Boolean expressions can also be combined using the operators

**and**
- **true** if both inputs are **true**;
- **false** otherwise

**or**
- **false** if both inputs are **false**;
- **true** otherwise
true and false
false
true or false
true
(1 < 2) and (2 > 3)
false
(1 <= 0) or (1 == 1)
true
To change an expression that evaluates to \texttt{true} to be \texttt{false} – or vice versa – use the \texttt{not} function:

```python
>>> not(true)
false
>>> not(1 == 0)
true
```
\[ i_1 = \text{rectangle}(10, 20, \text{"solid"}, \text{"red"}) \]
\[ i_2 = \text{rectangle}(20, 10, \text{"solid"}, \text{"blue"}) \]

\[ \text{image-width}(i_1) < \text{image-width}(i_2) \]
rect = rectangle(10, 20, "solid", "red")

if image-width(rect) < image-height(rect):
    "portrait"
else:
    "landscape"
end
if ... else ... end is a *conditional expression*. Conditionals allow us to *branch* – maybe we evaluate this expression, or maybe this other expression instead!
To form an if expression:

```plaintext
if ⟨expression⟩:
  ⟨expression⟩
else:
  ⟨expression⟩
end
```

- True–false question
- True ("then") answer
- False ("else") answer
How an if expression is evaluated

```python
if 1 < 2:
    "All is right in the world"
else:
    "Watch out for flying pigs"
end
```

1. If the question expression is not a value, evaluate it, and replace with the resulting value.
How an `if` expression is evaluated

```python
if true:
    "All is right in the world"
else:
    "Watch out for flying pigs"
end
```

1. If the question expression is not a value, evaluate it, and replace with the resulting value.
How an if expression is evaluated

if true:
   "All is right in the world"
else:
   "Watch out for flying pigs"
end

2 If the question is true, replace the entire if expression with the true ("then") answer expression.
How an if expression is evaluated

"All is right in the world"

2 If the question is true, replace the entire if expression with the true (“then”) answer expression.
How an if expression is evaluated

```python
if false:
    "All is right in the world"
else:
    "Watch out for flying pigs"
end
```

3 If the question is false, replace the entire if expression with the false (“else”) answer expression.
How an if expression is evaluated

"Watch out for flying pigs"

3 If the question is false, replace the entire if expression with the false (“else”) answer expression.
How an if expression is evaluated

```python
if 42:
    "All is right in the world"
else:
    "Watch out for flying pigs"
end
```

4 Otherwise, the question must be a value other than `true` or `false`, so produce an error.
How an *if* expression is evaluated

```python
if 42:
    "All is right in the world"
else:
    "Watch out for flying pigs"
end
```

4 Otherwise, the question must be a value other than `true` or `false`, so produce an error.
rect = rectangle(10, 20, "solid", "red")

if image-width(rect) < image-height(rect):
    "portrait"
else:
    "landscape"
end
rect = rectangle(10, 20, "solid", "red")

if image-width(rect) < image-height(rect):
    "portrait"
else if image-width(rect) == image-height(rect):
    "square"
else:
    "landscape"
end
\[
rect = \text{rectangle}(10, 20, \text{"solid"}, \text{"red"})
\]

fun \text{image-type}(img :: \text{Image}) -> \text{String}:
    doc: \"Classify an image as portrait, square, or landscape\"
    if \text{image-width}(img) < \text{image-height}(img):
        \"portrait\"
    else if \text{image-width}(img) == \text{image-height}(img):
        \"square\"
    else:
        \"landscape\"
    end
where:
    \text{image-type}(rect) is \"portrait\"
end
rect = rectangle(10, 20, "solid", "red")

fun image-type(img :: Image) -> String:
  doc: "Classify an image as portrait, square, or landscape"
  if image-width(img) < image-height(img):
    "portrait"
  else if image-width(img) == image-height(img):
    "square"
  else:
    "landscape"
end

where:
  image-type(rect) is "portrait"
  image-type(rectangle(10, 10, "solid", "blue")) is "square"
  image-type(rectangle(20, 10, "solid", "blue")) is "landscape"
Acknowledgments

This class incorporates material from:

Kathi Fisler, Brown University
Gregor Kiczales, University of British Columbia
Peter Lemieszewski, Vassar College