Problem Solving and Abstraction (CMPU 101)

Tom Ellman
Lecture 3
Data Types

- Number
- String
- Image
- Boolean

Named after George Boole, 19th century mathematician and logician.
Boolean Values

• There are:
  – Many numbers
  – Many strings
  – Many images

• But only two Boolean Values:
  
  true
  false
Operations on Boolean Values

Negation: `not`

```python
>>> not(True)
false

>>> not(False)
true
```
Negation Examples

```python
>>> obama-is-president = false
true

>>> not(obama-is-president)
true

>>> today-is-monday = true
false

>>> not(today-is-monday)
false
```
Operations on Boolean Values

Conjunction: `and`

```
>>> true and true
true
true

>>> true and false
false
false

>>> false and true
false and true
false

>>> false and false
false
false
```
Conjunction Examples

```python
>>> obama-is-president = False
>>> today-is-monday = True

>>> today-is-monday and obama-is-president
False

>>> today-is-monday and not(obama-is-president)
True

>>> not(today-is-monday) and not(obama-is-president)
False
```
Operations on Boolean Values

Disjunction: or

```python
>>> true or true
true

>>> true or false
true

>>> false or true
true

>>> false or false
false
```
Disjunction Examples

```python
>>> obama-is-president = false
>>> today-is-monday = true
>>> today-is-monday or obama-is-president
true
>>> not(today-is-monday) or not(obama-is-president)
true
>>> not(today-is-monday) or obama-is-president
false
```
Operations that Create Boolean Values

Equal: `==`

```python
>>> "foo" == "foo"
true
>>> "foo" == "bar"
false
>>> (2 + 4) == (4 + 2)
true
>>> (2 / 4) == (4 / 2)
false
```
Operations that Create Boolean Values

Less Than: <   Greater Than: >
Less or Equal: <=   Greater or Equal: >=

```
>>> 13 < 137
true
>>> 137 < 13
false
>>> 21 <= 42
true
>>> 21 <= 21
true
```

```
>>> "zebra" > "aardvark"
true
>>> "aardvark" < "zebra"
true
>>> "DAD" < "dad"
true
>>> "dad" < "DAD"
false
>>> "dad" < "dada"
true
```
Operations that Create Boolean Values
num-equal  string-equal   string-contains

```python
>>> num-equal(2, 1 + 1)
true
>>> string-equal("foo","bar")
false
>>> string-contains("foo","foobar")
false
>>> string-contains("foobar","foobar")
true
```

Why should one use `num-equal` or `string-equal` rather than `==` ?
AWD Surcharge

Determine the extra charge for all-wheel-drive (AWD) depending on the type of vehicle.

```haskell
sedan-awd-surcharge = 1000
suv-awd-surcharge = 2000

fun awd-surcharge(vehicle :: String) -> Number:
    # ...?...

where:
    awd-surcharge("sedan") is sedan-awd-surcharge
    awd-surcharge("suv") is suv-awd-surcharge

end
```

Why define constants? (So you can change prices in just one place in program.) Why put these definitions at the top? (Easy reference.)
Conditional Expression:

if ... else ... end

What part of the code handles SUVs? One must read the code above the else clause.
fun awd-surcharge1(vehicle :: String) -> Number:
    if (vehicle == "sedan"):
        sedan-awd-surcharge
    else:
        suv-awd-surcharge
    end
where:
    awd-surcharge1("sedan") is sedan-awd-surcharge
    awd-surcharge1("suv") is suv-awd-surcharge
end
Conditional Expression:
if ... else if ... end

In this type of conditional expression, we can put another if-clause in the else part of a conditional expression.
In this version we explicitly test whether the vehicle is suv after determining it’s not sedan. This second test is not need for the code to function correctly; however, the second test lets us see how suv is handled by looking only at the else-if clause.

Also, it’s easier to add more cases, like minivan ...
Handling Three or More Cases

```kotlin
fun awd-surcharge3(vehicle :: String) -> Number:
    if (vehicle == "sedan"):
        sedan-awd-surcharge
    else if (vehicle == "suv"):
        suv-awd-surcharge
    else if (vehicle == "minivan"):
        minivan-awd-surcharge
    end

where:
    awd-surcharge3("sedan") is sedan-awd-surcharge
    awd-surcharge3("suv") is suv-awd-surcharge
    awd-surcharge3("minivan") is minivan-awd-surcharge
end
```
## Computing Marginal Tax Rates

<table>
<thead>
<tr>
<th>lbd</th>
<th>ubd</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20,000</td>
<td>0</td>
</tr>
<tr>
<td>20,001</td>
<td>50,000</td>
<td>0.1</td>
</tr>
<tr>
<td>50,001</td>
<td>100,000</td>
<td>0.3</td>
</tr>
</tbody>
</table>
Computing Marginal Tax Rates

fun marginal-tax-rate1(income :: Number) -> Number:
    doc: "Marginal tax rate based on income."
    if income <= 20000:  0.0
    else if (income <= 50000): 0.1
    else if (income <= 100000): 0.3
    else: 1.0
end

where:
marginal-tax-rate1(15000) is 0.0
marginal-tax-rate1(20000) is 0.0
marginal-tax-rate1(35000) is 0.1
marginal-tax-rate1(50000) is 0.1
marginal-tax-rate1(70000) is 0.3
marginal-tax-rate1(100000) is 0.3
marginal-tax-rate1(125000) is 1.0
end

Notice that each else clause depends on clauses above it. This is concise but hard to read and understand.

Tests include boundary cases and cases in between boundaries.
Computing Marginal Tax Rates

fun marginal-tax-rate2(income :: Number) -> Number:
  doc: "Marginal tax rate based on income."
  if income <= 20000: 0.0
  else if (income > 20000) and (income <= 50000): 0.1
  else if (income > 50000) and (income <= 100000): 0.3
  else: 1.0
end

where:
  marginal-tax-rate2(15000) is 0.0
  marginal-tax-rate2(20000) is 0.0
  marginal-tax-rate2(35000) is 0.1
  marginal-tax-rate2(50000) is 0.1
  marginal-tax-rate2(70000) is 0.3
  marginal-tax-rate2(100000) is 0.3
  marginal-tax-rate2(125000) is 1.0
end

Notice that each else clause describes an income range in terms of upper and lower bounds – not depending on previous clauses. This is less concise, but easier to read and understand.
# Greeting One’s Co-Workers

<table>
<thead>
<tr>
<th>at-or-after</th>
<th>and-before</th>
<th>greeting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>“Working Late?”</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>“Good Morning!”</td>
</tr>
<tr>
<td>12</td>
<td>18</td>
<td>“Good Afternoon!”</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
<td>“Good Evening!”</td>
</tr>
</tbody>
</table>
fun greeting1(hour :: Number) -> String:
    if hour < 6: "Working Late?"
    else if hour < 12: "Good Morning"
    else if hour < 18: "Good Afternoon"
    else if hour < 24: "Good Evening"
end

where:
greeting1(0) is "Working Late?"
greeting1(3) is "Working Late?"
greeting1(6) is "Good Morning"
greeting1(8) is "Good Morning"
greeting1(12) is "Good Afternoon"
greeting1(16) is "Good Afternoon"
greeting1(18) is "Good Evening"
greeting1(22) is "Good Evening"
Greeting One’s Co-Workers

fun greeting2(hour :: Number) -> String:
  if hour <= 5: "Working Late?"
  else if hour <= 11: "Good Morning"
  else if hour <= 17: "Good Afternoon"
  else if hour <= 23: "Good Evening"
end

where:
  greeting1(0) is "Working Late?"
  greeting1(3) is "Working Late?"
  greeting1(6) is "Good Morning"
  greeting1(8) is "Good Morning"
  greeting1(12) is "Good Afternoon"
  greeting1(16) is "Good Afternoon"
  greeting1(18) is "Good Evening"
  greeting1(22 ) is "Good Evening"
Greeting One’s Co-Workers

fun greeting3(hour :: Number) -> String:
    if (hour >= 0) and (hour < 6): "Working Late?"
    else if (hour >= 6) and (hour < 12): "Good Morning"
    else if (hour >= 12) and (hour < 18): "Good Afternoon"
    else if (hour >= 18) and (hour < 24): "Good Evening"
end

where:
    greeting2(0) is "Working Late?"
    greeting2(3) is "Working Late?"
    greeting2(6) is "Good Morning"
    greeting2(8) is "Good Morning"
    greeting2(12) is "Good Afternoon"
    greeting2(16) is "Good Afternoon"
    greeting2(18) is "Good Evening"
    greeting2(22) is "Good Evening"
end
Find the Maximum of Three Numbers

```plaintext
fun maximum1(a :: Number, b :: Number, c :: Number) -> Number:
  ...
where:
  maximum1(3,2,1) is 3
  maximum1(3,4,5) is 5
  maximum1(3,9,6) is 9
end
```
Maximum of Three Numbers (Version 1)

In each of the three cases, we compare one value to each of the other two values. Is this really necessary? Can we make it simpler?

```haskell
fun maximum1(a :: Number, b :: Number, c :: Number) -> Number:
    if (a >= b) and (a >= c): a
    else if (b >= a) and (b >= c): b
    else if (c >= a) and (c >= b): c
end

where:
    maximum1(3,2,1) is 3
    maximum1(3,4,5) is 5
    maximum1(3,9,6) is 9
end
```
Maximum of Three Numbers (Version 2)

```haskell
fun maximum2(a :: Number, b :: Number, c :: Number) -> Number:
    if (a >= b) and (a >= c): a
    else if (b >= c): b
    else: c
end
where:
    maximum2(3,2,1) is 3
    maximum2(3,4,5) is 5
    maximum2(3,9,6) is 9
end
```

After we eliminate \texttt{a} as maximum, we need not compare \texttt{b} to \texttt{a}. After we’ve eliminated both \texttt{a} and \texttt{b}, we know that \texttt{c} is maximum without doing any more comparisons.
Rock Paper Scissors

• Rock smashes scissors

• Scissors cuts paper

• Paper wraps rock.

• All other cases are a tie.
Rock Paper Scissors (Version 1)

- Rock and rock tie.
- Rock smashes scissors.
- Paper wraps rock.
Rock Paper Scissors (Version 1)

```kotlin
fun rsp1(alice :: String, bob :: String) -> String:
    if (alice == bob): "tie"
    else if (alice == "rock") and (bob == "scissors"): "alice"
    else if (alice == "scissors") and (bob == "paper"): "alice"
    else if (alice == "paper") and (bob == "rock"): "alice"
    else: "bob"
end

where:
    rsp1("rock","rock") is "tie"
    rsp1("rock","scissors") is "alice"
    rsp1("rock","paper") is "bob"
end
```

After checking for a tie, we explicitly check each of the ways that **alice** wins. If none of them apply, then **bob** must win.
Rock Paper Scissors (Version 2)

fun rsp2(alice :: String, bob :: String) -> String:
    if (alice == bob): "tie"
    else if (alice == "rock") and (bob == "scissors"): "alice"
    else if (alice == "scissors") and (bob == "paper"): "alice"
    else if (alice == "paper") and (bob == "rock"): "alice"
    else if (bob == "rock") and (alice == "scissors"): "bob"
    else if (bob == "scissors") and (alice == "paper"): "bob"
    else if (bob == "paper") and (alice == "rock"): "bob"
end

where:
    rsp2("rock","rock") is "tie"
    rsp2("rock","scissors") is "alice"
    rsp2("rock","paper") is "bob"
end

Here we explicitly check each of the ways that alice wins and all the ways that bob wins. The code takes longer to write, but is perhaps easier to understand.

How many test cases do we need to consider all possibilities?