Functional Programming for Logicians

Péter Mekis

Department of Logic, ELTE Budapest

Session 2: 2019 February 18

Static vs dynamic typing

Python

```
def foo(s):
    if s == " ":  return 1
    else:  return "1"
print(foo(input()))
```

Static vs dynamic typing

Python

```
def foo(s):
    if s == " ":  return 1
    else:  return "1"
print(foo(input()))
```

Runs without error for any input.

Static vs dynamic typing

Python

```
def foo(s):
    if s == " ":  return 1
    else:  return "1"
print(foo(input()))
```

Runs without error for any input.

Haskell

```
foo ::  String -> a
foo x
  | x == " " = 1
  | otherwise = "1"
```

Static vs dynamic typing

Python

```
def foo(s):
    if s == " ":  return 1
    else:  return "1"
print(foo(input()))
```

Runs without error for any input.

Haskell

```
foo ::  String -> a
foo x
  | x == " " = 1
  | otherwise = "1"
```

Doesn't compile.

Strong vs weak typing

Haskell

```
foo ::  a -> Int
foo x = x + 1
```

Strong vs weak typing

Haskell

```
foo ::  a -> Int
foo x = x + 1
```

Doesn't compile.

Strong vs weak typing

Haskell

```
foo ::  a -> Int
foo x = x + 1
```

Doesn't compile.

Python

```
def foo(s):
    if s == " ":  return 1
    else:  return "1"
print(foo(input()) + 1)
```

Strong vs weak typing

Haskell

```
foo ::  a -> Int
foo x = x + 1
```

Doesn't compile.

Python

```
def foo(s):
    if s == " ":  return 1
    else:  return "1"
print(foo(input()) + 1)
```

Any input other than " " raises a runtime error.

Strong vs weak typing

Haskell

```
foo ::  a -> Int
foo x = x + 1
```

Doesn't compile.

Python

```
def foo(s):
    if s == " ":  return 1
    else:  return "1"
print(foo(input()) + 1)
```

Any input other than " " raises a runtime error.

JavaScript

```
function foo(x) {
  if (x == " ") {return 1}
  else {return "1"}
}
document.writeln(foo(" ") + 1);
document.writeln(foo("@") + 1);
```

Strong vs weak typing

Haskell

```
foo ::  a -> Int
foo x = x + 1
```

Doesn't compile.

Python

```
def foo(s):
    if s == " ":  return 1
    else:  return "1"
print(foo(input()) + 1)
```

Any input other than " " raises a runtime error.

JavaScript

```
function foo(x) {
  if (x == " ") {return 1}
  else {return "1"}
}
document.writeln(foo(" ") + 1);
document.writeln(foo("@") + 1);
```

No error is raised.

Recursion

- "A journey of a thousand miles begins with a single step."

Recursion

- "A journey of a thousand miles begins with a single step."
- Laozi's approach to a journey:

```
take_journ (pres_loc) (dest):
if pres_loc == dest:  stay(pres_loc)
else:  take_journ (take_a_step(pres_loc)) (dest)
```

Recursion

- "A journey of a thousand miles begins with a single step."
- Laozi's approach to a journey:

```
take_journ (pres_loc) (dest):
if pres_loc == dest:  stay(pres_loc)
else:  take_journ (take_a_step(pres_loc)) (dest)
```

- Structure:
  - *outer function* to be defined: `take_journ`
  - *inner function* used in the definition: `take_a_step`: when applying `take_a_step`, `take_journ` calls itself (*recurs*)
  - *base case*: `pres_loc == dest` – the return value is given without without calling `take_journ`

Recursion

- "A journey of a thousand miles begins with a single step."
- Laozi's approach to a journey:

```
take_journ (pres_loc) (dest):
if pres_loc == dest:  stay(pres_loc)
else:  take_journ (take_a_step(pres_loc)) (dest)
```

- Structure:
  - *outer function* to be defined: `take_journ`
  - *inner function* used in the definition: `take_a_step`: when applying `take_a_step`, `take_journ` calls itself (*recurs*)
  - *base case*: `pres_loc == dest` – the return value is given without without calling `take_journ`
- Addition:
  - outer function: `(+)`
  - inner function: `succ`
  - base case: `n + 0 = n`

List: a recursive type class

- Processing a list:
  - outer function: process the list
  - inner function: read the head
  - base case: empty list

List: a recursive type class

- Processing a list:
  - outer function: process the list
  - inner function: read the head
  - base case: empty list
- Haskell's definition of lists:
  - `data [] a = [] | a : [a]`
  - `[0,1,2] == 0:(1:(2:[]))`
  - `"Donald" == 'D':('o':('n':('a':('l':('d':[])))))`

List: a recursive type class

- Processing a list:
  - outer function: process the list
  - inner function: read the head
  - base case: empty list
- Haskell's definition of lists:
  - `data [] a = [] | a : [a]`
  - `[0,1,2] == 0:(1:(2:[]))`
  - `"Donald" == 'D':('o':('n':('a':('l':('d':[])))))`
- Cf. definition of tuples in set theory:
  - $(a, b) = \{\{a\}, \{a, b\}\}$
  - $(a_0, \ldots, a_n) = (a_0, (a_1, \ldots, q_n))$

List: a recursive type class

- Processing a list:
  - outer function: process the list
  - inner function: read the head
  - base case: empty list
- Haskell's definition of lists:
  - `data [] a = [] | a : [a]`
  - `[0,1,2] == 0:(1:(2:[]))`
  - `"Donald" == 'D':('o':('n':('a':('l':('d':[])))))`
- Cf. definition of tuples in set theory:
  - $(a, b) = \{\{a\}, \{a, b\}\}$
  - $(a_0, \ldots, a_n) = (a_0, (a_1, \ldots, q_n))$
  - Wouldn't work with haskell's List class, items have different types.

List: a recursive type class

- Processing a list:
  - outer function: process the list
  - inner function: read the head
  - base case: empty list
- Haskell's definition of lists:
  - `data [] a = [] | a : [a]`
  - `[0,1,2] == 0:(1:(2:[]))`
  - `"Donald" == 'D':('o':('n':('a':('l':('d':[])))))`
- Cf. definition of tuples in set theory:
  - $(a, b) = \{\{a\}, \{a, b\}\}$
  - $(a_0, \ldots, a_n) = (a_0, (a_1, \ldots, q_n))$
  - Wouldn't work with haskell's List class, items have different types.
  - Pairs as primitives in typed set theories and NF work similarly as Haskell's list constructor.

List: a recursive type class

- Processing a list:
  - outer function: process the list
  - inner function: read the head
  - base case: empty list
- Haskell's definition of lists:
  - `data [] a = [] | a : [a]`
  - `[0,1,2] == 0:(1:(2:[]))`
  - `"Donald" == 'D':('o':('n':('a':('l':('d':[])))))`
- Cf. definition of tuples in set theory:
  - $(a, b) = \{\{a\}, \{a, b\}\}$
  - $(a_0, \ldots, a_n) = (a_0, (a_1, \ldots, q_n))$
  - Wouldn't work with haskell's List class, items have different types.
  - Pairs as primitives in typed set theories and NF work similarly as Haskell's list constructor.
- Common list operations:
  - `head(s) :: [a] -> a`

List: a recursive type class

- Processing a list:
  - outer function: process the list
  - inner function: read the head
  - base case: empty list
- Haskell's definition of lists:
  - `data [] a = [] | a : [a]`
  - `[0,1,2] == 0:(1:(2:[]))`
  - `"Donald" == 'D':('o':('n':('a':('l':('d':[])))))`
- Cf. definition of tuples in set theory:
  - $(a, b) = \{\{a\}, \{a, b\}\}$
  - $(a_0, \ldots, a_n) = (a_0, (a_1, \ldots, q_n))$
  - Wouldn't work with haskell's List class, items have different types.
  - Pairs as primitives in typed set theories and NF work similarly as Haskell's list constructor.
- Common list operations:
  - `head(s) :: [a] -> a`
  - `tail(s) :: [a] -> [a]`

List: a recursive type class

- Processing a list:
  - outer function: process the list
  - inner function: read the head
  - base case: empty list
- Haskell's definition of lists:
  - `data [] a = [] | a : [a]`
  - `[0,1,2] == 0:(1:(2:[]))`
  - `"Donald" == 'D':('o':('n':('a':('l':('d':[])))))`
- Cf. definition of tuples in set theory:
  - $(a, b) = \{\{a\}, \{a, b\}\}$
  - $(a_0, \ldots, a_n) = (a_0, (a_1, \ldots, q_n))$
  - Wouldn't work with haskell's List class, items have different types.
  - Pairs as primitives in typed set theories and NF work similarly as Haskell's list constructor.
- Common list operations:
  - `head(s) :: [a] -> a`
  - `tail(s) :: [a] -> [a]`
  - `(++) :: [a] -> [a] -> [a]`

Case selection

- if–then–else:

```
nplus :: Int -> Int -> Int
nplus n m = if m == 0 then n else succ (nplus n (pred m))
```

Case selection

- if–then–else:

```
nplus :: Int -> Int -> Int
nplus n m = if m == 0 then n else succ (nplus n (pred m))
```

- pattern matching:

```
nplus’ :: Int -> Int -> Int
nplus’ n 0 = n
nplus’ n m = succ (nplus n (pred m))
```

Case selection

- if–then–else:

```
nplus :: Int -> Int -> Int
nplus n m = if m == 0 then n else succ (nplus n (pred m))
```

- pattern matching:

```
nplus' :: Int -> Int -> Int
nplus' n 0 = n
nplus' n m = succ (nplus n (pred m))
```

- guards:

```
nplus'' :: Int -> Int -> Int
nplus'' n m
  | m == 0    = n
  | otherwise = succ (nplus n (pred m))
```