Functional Programming

Introduction

H. Turgut Uyar

2013-2016

License

Topics

Paradigms

- paradigm: approach to programming
- based on a set of principles or theory
- different paradigms: different ways of thinking
- idioms: patterns for using language features

Paradigms

- imperative: how to solve
- procedural, object-oriented

- declarative: what to solve
- functional, logic

Universality

- universal: capable of expressing any computation
- any language that supports iteration or recursion is universal

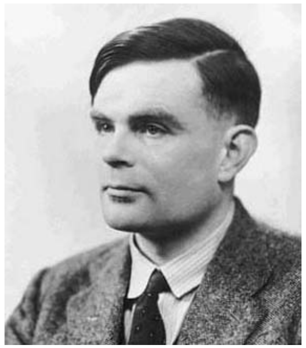
- Church-Turing thesis:

  Any real-world computation can be translated
  into an equivalent computation involving a Turing machine.

  It can also be calculated using general recursive functions.

  (`http://mathworld.wolfram.com/`)

Imperative Programming



Alan Turing
(1912-1954)

- based on the Turing machine
- program: sequence of instructions
  for a von Neumann computer
- contents of memory constitute state
- statements update variables
  (mutation)
- assignment, control structures
- natural model of hardware

Imperative Programming Example

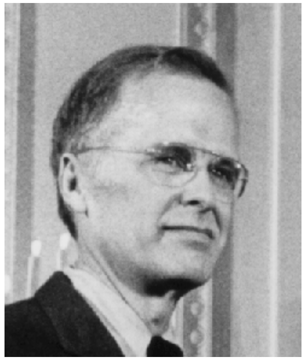
greatest common divisor (Python)

```
def gcd(x, y):
    r = 0
    while y > 0:
        r = x % y
        x = y
        y = r
    return x
```

x	y	r
9702	945	0
945	252	252
252	189	189
189	63	63
63	0	0

~> 63

Milestones in Imperative Programming Languages



John Backus
(1924-2007)

- Fortran (1957)
- ALGOL (1960)
- C (1972)
- Ada (1983)
- Java (1995)

Functional Programming

Alonzo Church
(1903-1995)

- based on λ-calculus
- program: function application
- same inputs should produce same output ("pure")
- function modifies context $\rightarrow$ side effect
- avoid mutation
- higher-order functions

Functional Programming Example

greatest common divisor (Python)

```
def gcd(x, y):
    if y == 0:
        return x
    else:
        return gcd(y, x % y)
```

```
gcd(9702, 945)
~> gcd(945, 252)
   ~> gcd(252, 189)
      ~> gcd(189, 63)
         ~> gcd(63, 0)
            ~> 63
         ~> 63
      ~> 63
   ~> 63
~> 63
```


Side Effects

- sources of side effects: global variables

example

```
factor = 0

def multiples(n):
    global factor
    factor = factor + 1
    return factor * n
```

Side Effects

- sources of side effects: function state, object state

example

```
class Multiplier:
    def __init__(self):
        self.factor = 0

    def multiples(self, n):
        self.factor = self.factor + 1
        return self.factor * n
```


Side Effects

- sources of side effects: input/output

example

```
def read_byte(f):
    return f.read(1)
```


Side Effects

- sources of side effects: randomness

example

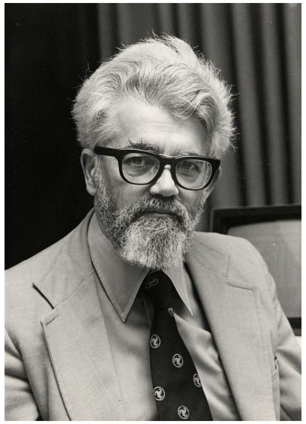
```
import random

def get_random(n):
    return random.randrange(1, n + 1)
```


Problems with Side Effects

- harder to reason about programs
- harder to test programs
- harder to parallelize programs

- could we write programs without side effects?
- or, at least, could we separate pure and impure parts?

Milestones in Functional Programming Languages



John McCarthy
(1927-2011)

- Lisp (1957)
- ML (1973)
- Haskell (1990)

Multiple Paradigms

- functional languages with object-oriented features
- Ocaml, F#, Scala

- imperative languages with functional features
- Python, Ruby, C#, Java

- what makes a language functional or imperative?
- higher-order functions
- immutable data structures
- recommended idioms in functional style

Expressions and Statements

- an expression is evaluated to produce a value
- a statement is executed to update a variable

Expression and Statement Example

- conditional statement (Python)

```
if x < 0:
    abs_x = -x
else:
    abs_x = x
```

- conditional expression (Python)

```
abs_x = -x if x < 0 else x
```

- conditional expression (Haskell)

```
abs_x = if x < 0 then -x else x
```

Expression and Statement Example

- bad:

```
if age < 18:
    minor = True
else:
    minor = False
```

- better:

```
minor = True if age < 18 else False
```

- much better:

```
minor = age < 18
```


Definitions

- binding: an association between an identifier and an entity
- environment: a set of bindings
- signature: name, type
- definition: name, expression

```
n :: t
n = e
```

- redefining not allowed

Definition Examples

```
-- diameter of the circle
d :: Float
d = 4.8

-- circumference of the circle
c :: Float
c = 3.14159 * d

-- d = 15.62 ~> error: multiple declarations
```


Local Definitions

- local definition: used only within expression

```
n = e
  where
    n1 :: t1
    n1 = e1

    n2 :: t2
    n2 = e2

    ...
```

```
let
    n1 :: t1
    n1 = e1

    n2 :: t2
    n2 = e2

    ...
in
  n = e
```

Local Definition Example

```
-- diameter of the circle
d :: Float
d = 4.8

-- area of the circle
a :: Float
a = 3.14159 * r * r
  where
    r :: Float
    r = d / 2.0
```


Type Inference

- Haskell can infer types (more on that later)
- we will leave out type declarations for data in local definitions

example

```
a :: Float
a = 3.14159 * r * r
  where
    r = d / 2.0
```


Functions

- imperative: function body is a block
- special construct for sending back the result: `return`

- functional: function body is an expression

Function Definitions

- function definition:

```
n :: t1 -> t2 -> ... -> tk -> t
n x1 x2 ... xk = e
```

- function application:

```
n e1 e2 ... ek
```

Function Examples

```
sqr :: Integer -> Integer
sqr x = x * x

-- sqr 21      ~> 441
-- sqr (2 + 5) ~> 49
-- sqr -2      ~> error
-- sqr (-2)    ~> 4

sumOfSquares :: Integer -> Integer -> Integer
sumOfSquares x y = sqr x + sqr y

-- sumOfSquares 3 4       ~> 25
-- sumOfSquares 2 (sqr 3) ~> 85
```


Function Example

```
sumOfCubes :: Integer -> Integer -> Integer
sumOfCubes x y = cube x + cube y
  where
    cube :: Integer -> Integer
    cube n = n * n * n
```


Infix - Prefix

- functions infix when in backquotes

```
mod n 2
n `mod` 2
```

- operators prefix when in parentheses

```
6 * 7
(*) 6 7
```


Guards

- writing conditional expressions can become complicated
- guards: predicates to check cases

```
n :: t1 -> t2 -> ... -> tk -> t
n x1 x2 ... xk
  | p1        = e1
  | p2        = e2
    ...
  | otherwise = e
```

- function result is the expression for the first satisfied predicate

Guard Example

maximum of three integers

```
maxThree :: Integer -> Integer -> Integer -> Integer
maxThree x y z
  | x>=y && x>=z = x
  | y>=z         = y
  | otherwise    = z
```


Errors

- errors can be reported using `error`
- doesn't change the type signature

example: reciprocal (multiplicative inverse)

```
reciprocal :: Float -> Float
reciprocal x
  | x == 0    = error "division by zero"
  | otherwise = 1.0 / x
```


References

Required Reading: Thompson

- Chapter 1: Introducing functional programming
- Chapter 2: Getting started with Haskell and GHCi