Functional Programming Language

Nguyen Hua Phung, Ph.D.
Dept. of CSE, HCMUT
Outline

- History
- Overview
- Main features
- Haskell
History

- Late 1950s, LISP (List Processing) by John McCarthy
  - general list structures
  - function application

- 1960s, ISWIM by Peter Landin
  - completely based on mathematical formalisms
  - its behavior was described with complete precision
History

- 1970s and 1980s,
  - Scheme
    - more uniform than LISP
    - resemble more closely the lambda calculus
  - Common LISP
    - a standard for LISP family
  - ML, Miranda
    - syntax more closely related to Pascal
    - type checking similar to Pascal
History

- 1990s-present, Haskell
  - purely functional language
  - fully-curried functions
  - lazy evaluation
  - function overloading
Overview

- a computation $\rightarrow$ a mathematical function mapping inputs to outputs
  
  $f: X \rightarrow Y$

  $x \mapsto y = f(x)$

  - uniform view of programs as functions
  - treatment of functions as data
  - limitation of side effects (no assignments, loops)

  $\Rightarrow$ tools for prototyping

  $\Rightarrow$ artificial intelligence

  $\Rightarrow$ mathematical proof systems
Overview

Why have functional languages never become “main-stream” ones?
- inefficiency of execution
- difficult to study
- studied after imperative or object-oriented languages
Outline

- History
- Overview
- **Main features**
- Haskell
Function definition

\[ f : \mathcal{X} \rightarrow \mathcal{Y} \]

- Function as a set of pairs
  \[
  f \equiv \{(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(8,9),(9,0)\}
  \]
- Definition by comprehension (formula)
  \[
  f(x) = (x + 1) \mod 10
  \]
  - Recursive definition
    \[
    fact(n) = \text{if } (n=0) \text{ then } 1 \text{ else } n \ast fact(n-1)
    \]
Lambda calculus

- invented by Alonzo Church
- a mathematical formalism for expressing computation by functions
- lambda calculus is equivalent to Turing machines
Lambda abstraction

- \( \text{exp} \rightarrow \text{constant} \mid \text{variable} \mid (\text{exp} \ \text{exp}) \mid (\lambda \ \text{variable}. \ \text{exp}) \)

- \( (\lambda \ x. \ (+ \ 1) \ x) \) ⇒ an unnamed function of parameter \( x \) that adds 1 to \( x \)

- \( ((\lambda \ x. \ (+ \ 1) \ x) \ 2) \) ⇒ application of expressions
  \[ \Rightarrow ((+ \ 1) \ 2) \Rightarrow 3 \]

- \( \text{free}(x) = \{x\} \)
- \( \text{free}(M \ N) = \text{free}(M) \cup \text{free}(N) \)
- \( \text{free}(\lambda x. E) = \text{free}(E) - \{x\} \)
- \( \text{bound}(E) = \{x | x \text{ appears in } E \text{ and } x \notin \text{free}(E)\} \)

\( (\lambda \ x. * \ x \ y) \) ⇒ \( x \) is bound and \( y \) is free
Transforming expressions

- **Beta-conversion**: \((\lambda x. E) F \equiv E[F/x]\) where \(E[F/x]\) is \(E\) with all free occurrences of \(x\) in \(E\) replaced by \(F\)
  
  \[
  (\lambda x. + 1 x) 2 \Rightarrow (+ 1 2) \Rightarrow 3
  \]
  
  \[
  (\lambda x. + ((\lambda y.((\lambda x.* x y) 2)) x) y) \Rightarrow ?
  \]
  
  \[
  ((\lambda x.\lambda y.((+ x) y)) y) \not\equiv (\lambda y. + y y)
  \text{ name capture problem}
  \]

- **Alpha-conversion**: \((\lambda x. E) \equiv (\lambda y. E[y/x])\)
  
  \[
  (\lambda y. + x y) \Rightarrow (\lambda z. + x z)
  \]

- **Eta-conversion**: \((\lambda x. (E x)) \equiv E\) if \(E\) contains no free occurrences of \(x\)
  
  \[
  (\lambda x.\lambda y.((+ x) y))) \Rightarrow (\lambda x. (+ x)) \Rightarrow +
  \]
Order of evaluation

- **Applicative order evaluation**
  \[ ((\lambda x. x x) (+ 2 3)) \Rightarrow ((\lambda x. x x) 5) \Rightarrow (* 5 5) \Rightarrow 25 \]

- **Normal order evaluation**
  \[ ((\lambda y. 2) ((\lambda x. x x) (\lambda x. x x))) \Rightarrow ? \]
  \[ ((\lambda x. x x) (+ 2 3)) \Rightarrow (* (+ 2 3) (+ 2 3)) \Rightarrow (* 5 5) \Rightarrow 25 \]
Outline

- History
- Overview
- Main features
- Haskell
### Expressions

#### Infix

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Left-Associative</th>
<th>Non-Associative</th>
<th>Right-Associative</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>!, !!, //</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>**, ^, ^^</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>* , /, div, mod</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>+, -</td>
<td>:+</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>\</td>
<td>. , ++</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>/=, &lt;, &lt;=, ==, &gt;, &gt;=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>&gt;&gt;, &gt;&gt;=</td>
<td>:=</td>
<td></td>
</tr>
</tbody>
</table>

2+2  
5 * (4 + 6) - 2

#### Prefix

(-) ((*) 5 ((+ 4 6)) 2
Elementary Types and Values

- **Bool**: True, False
- **Int**: $-2^{31}$ to $2^{31}-1$
- **Integer**: integers of any size
- **Char**: ‘a’
- **String**: “abc” [‘a’, ‘b’, ‘c’]
- **Float**: 3.14 0.0314e4
Lists

List definitions
- evens = [0, 2, 4, 6, 8]
- odds = [1, 3..9]
- moreevens = [ 2*x | x <- [0..10]]
- mostevens = [ 2*x | x <- [0,1..]]
- mostevens = [0,2..]

List operators
- 8:[] -- gives [8]
- 6:8:[] -- gives [6,8]
- 4:[6,8] -- gives [4,6,8]
- head evens -- gives 0
- tail evens -- gives [2,4,6,8]
- [1,2]++[3,4] -- gives [1,2,3,4]
- null [] -- gives True
- null evens -- gives False
- [1,2]==[1,2] -- gives True
- [1,2]==[2,1] -- gives False
Tuples

- type Entry = (Person, Number)
- type Person = String
- type Number = Integer
- type PhoneBook = [Entry]

find PhoneBook -> Person -> [Number]
find pb p = [ n | (person, n) <- pb, person == p]

pb = [('an', 8345678), ('binh', 4321123), ('an', 0901111111)]
find pb "an" -- gives [8345678, 0901111111]
User-defined Types

```haskell
data BST a = Nil | Node a (BST a) (BST a)

flatten :: BST a -> [a]
flatten Nil = []
flatten (Node val left right) =
    (flatten left) ++ [val] ++ (flatten right)

flatten (Node 1 (Node 2 Nil Nil) Nil) -- gives ???
```
Control Flow

- if..then..else
  
  if \( x \geq y \) \&\& \( x \geq z \) then \( x \)
  
  else if \( y \geq x \) \&\& \( y \geq z \) then \( y \)
  
  else \( z \)

- \( x \geq y \) \&\& \( x \geq z \) = \( x \)

- \( y \geq x \) \&\& \( y \geq z \) = \( y \)

- otherwise = \( z \)
Function definitions

- name :: Domain -> Range
  name parameters
    | g1 = e1
    | g2 = e2
    ...
    | otherwise = e

- max3 :: Int -> Int -> Int -> Int
  max3 x y z
    | x >= y && x >= z = x
    | y >= x && y >= z = y
    | otherwise = z

Curried function
max_100 = max3 100
Example 1

to compute the list of prime number (the sieve of Eratosthenes)

\[ [2,3,4,5,6,7,8,9,10,11] \Rightarrow [2,3,5,7,9,11] \text{ (cancel multiples of 2)} \]
\[ \Rightarrow [2,3,5,7,11] \text{ (cancel multiples of 3)} \]

\[ \text{sieve}(p:\text{lis}) = p : \text{sieve} [ n \mid n \leftarrow \text{lis}, \mod n p \neq 0 ] \]

\[ \text{primes} = \text{sieve}[2..] \]

normal order evaluation (lazy evaluation)
Example 2

- **take n first elements of a list**
  
  \[
  \text{take } 0 \ _ \ = \ [] \\
  \text{take } \_ \ [\] \ = \ [] \\
  \text{take } n \ (h:t) \ = \ h: \text{take} \ (n-1) \ t
  \]

  take 100 primes

- **drop n first elements of a list**
Example 3

- $(\lambda x \rightarrow x \times x) \ 3 \quad \text{-- gives 9}$  
- $\text{map :: (a -> b) -> [a] -> [b]}$
- $\text{map} \ (\lambda x \rightarrow x \times x) \ [1,2,3] \quad \text{-- gives [1,4,9]}$
- $\text{square_list} = \text{map} \ (\lambda x \rightarrow x \times x)$
- $\text{square_list} \ [1,2,3] \quad \text{-- gives [1,4,9]}$