Part 2
Functional programming in Haskell

Functional programming

- early functional flavored languages: LISP (John McCarthy, late 1950s), then IPL and APL
- 1977: John Backus Turing Award lecture “Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs.”
- 1970: ML (Robin Milner, University of Edinburgh)
- several ML dialects, most common now Objective Caml and Standard ML
- 1970s: Scheme (Lisp dialect) brought functional programming to the wider programming-languages community
- following Miranda (David Turner, 1985), interest in lazy functional languages grew: by 1987, more than a dozen
- at FPCA ’87 in Portland, consensus that a committee should define an open standard for such languages
- first version defined in 1990
- Haskell 98: stable, minimal, portable version of the language with standard library for teaching, and as a base for future extensions
- in January 2003 revised version
- Glasgow Haskell Compiler (GHC) current de facto standard implementation
- from 2006, ongoing process of defining a successor to the Haskell 98 standard (last revision published in July 2010)

Basics: lambda-expressions

- lambda-calculus forms the basis, as in almost all functional programming languages today
- expressions which denote functions: `\ x -> x+1`
- function application `(\ x -> x+1) 2`
- conventions like in the lambda calculis
  - $t_1\ t_2\ t_3 = (t_1\ t_2)\ t_3$
  - $\lambda x.t_1\, t_2 = \lambda x.(t_1\ t_2)$

- declarations of functions:

```
inc = \x -> x+1
inc x = x + 1
```


Basics: types and declarations

- each value has a type, the following are *type signature declarations*

```
5 :: Integer
'a' :: Char
\x -> x + 1 :: Integer -> Integer
[1,2,3] :: [Integer]
('b',4) :: (Char,Integer)
```

- the type system is sound, and *infers* type signatures

```
:type "elena"
"elena" :: [Char]
```

- types universally quantified over all types, e.g.,
- $\forall a\ [a]$ is the type of all homogeneous lists
- quantifier is omitted

Functions

- Higher-order functions

```
double f x = f (f x)

compose (f, g) x = f (g x)

compose (f, g) = \x -> f (g x)

compose = \(f, g) -> \x -> f (g x)

*Main> compose (inc, inc) 1
3
*Main> double inc 5
7
```


Functions

- Curried functions

```
sum:: (Integer, Integer) -> Integer
sum (x,y) = x + y
sum(1,2)

add:: Integer -> Integer -> Integer
add x y = x + y

*Main> add 1 2
3

compose f g x = f(g x)
```

- Partial application

```
inc = add 1

*Main> :type compose inc inc

compose inc inc :: Integer -> Integer

*Main> :type compose inc

compose inc :: (t -> Integer) -> t -> Integer
```

- given a function $f\colon A \times B \to C$, its curried version $\tilde{f}\colon A \to B \to C$ is defined by: for all $a \in A$,

  $\tilde{f}(a)\colon B \to C$,
  for all $b \in B$, $\tilde{f}(a)(b) = f(a, b)$

- conversely, given a function $g\colon A \to B \to C$, its uncurried version $\hat{g}\colon A \times B \to C$ is defined by: for all $a \in A$, $b \in B$,

  $\hat{g}(a, b) = g(a)(b)$

- curry and uncurry operators can be defined in Haskell:

```
curry f = \a -> \b -> f (a, b)

uncurry g = \(a, b) -> g a b

*Main> :type curry sum

curry sum :: Integer -> Integer -> Integer

*Main> :type curry

curry :: ((a, b) -> c) -> a -> b -> c
```


Polymorphism

- the definition of the identity function $f(x) = x$ makes sense independently from the nature of the argument
- in languages allowing polymorphism it is possible to write such definitions: `\x -> x`
- *one* definition applicable to arguments of different types
- different from *overloading*: same name for different definitions

```
*Main> :type (\x -> x)
(\x -> x) :: t -> t
*Main> :type compose
compose :: (t1 -> t2) -> (t -> t1) -> t -> t2
first(x,y) = x
*Main> :type (first)
(first) :: (t, t1) -> t
```

Polymorphism

- some types are *more general* than others, e.g., `[a] -> a` is more general than `[Integer] -> Integer`
- any expression has a *most general* or *principal* type
- the principal type represents all the different types a function can assume
- the type of `compose` in the expression `compose inc inc` is

  ```
  (Integer->Integer) -> (Integer->Integer)->Integer->Integer
  ```

  obtained by instantiating the type variables
- each (well-typed) Haskell expression has a unique principal (most general) type
- type inference: the programmer is not required to insert type annotations

Functions

- infix operators are just functions and can be defined:

  ```
  (++):: [a] -> [a] -> [a]
  []++xs = xs
  (x:xs)++ys = x:(xs++ys)

  (.) :: (b -> c) -> (a -> b) -> (a -> c)
  f.g = \x -> f(g x)
  ```

- partial applications of infix operators are called sections

  ```
  (x+) ≡ \y -> x + y
  (+y) ≡ \x -> x + y
  (+) ≡ \x y -> x + y
  ```

Pattern matching

- general form

```
f p1 = e1
     ...
f pn = en
```

- pattern = expression with free variables, describing a possible shape of the argument
- patterns are considered in the given order, hence each pattern behaves like a filter for the following (unless irrefutable)
- example

```
negate True = False
negate False = True
```

or

```
negate True = False
negate x = True
```

or using a wild-card

```
negate True = False
negate _ = True
```


- an exception is raised if a function is invoked on an argument which does not match any pattern:

```
*Main> let f 0 = 0 in f 1
*** Exception: <interactive>:1:4-10: Non-exhaustive
 patterns in function f
```

Another example

Implication

```
implies True False = False
implies _ _ = True
```

- a variable cannot be repeated, e.g.:

```
*Main> let f x x = 0 in f 0 0
<interactive>:1:6:
    Conflicting definitions for ‘x’
    Bound at: <interactive>:1:6
              <interactive>:1:8
    In the definition of ‘f’
```


Lists

- `[1,2,3]` is a shorthand for `1:2:3:[]`
- example of function defined by pattern-matching:

```
length [] = 0
length (_:xs) = 1 + length xs
```

- is a polymorphic function

```
length:: [a] -> Integer
length [1,2,3]
length ['a','b','c']
length [[1],[2,3],[4,5,6]]
```

- other polymorphic functions:

```
head:: [a] -> a
head (x:_) = x
tail::[a]->[a]
tail (_:xs) = xs
```

Polymorphic functions on lists

```
map :: (t -> a) -> [t] -> [a]
map f [] = []
map f (x:xs) = f x : map f xs

map (\x->x+1) [1,2,3,4]
[2,3,4,5]

itlist :: (t1 -> t -> t1) -> t1 -> [t] -> t1
itlist f a [] = a
itlist f a (x:xs) = itlist f (f a x) xs

sumlist = itlist (+) 0

flatten = itlist (++) []

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = (if p x then [x] else [])++(filter p xs)
filter (\x->x>5)[1,2,3,4,5,6,7]
[6,7]
```


List comprehension

```
filter p xs = [x | x <- xs, p x]

quicksort [] = []
quicksort (x:xs) =
  quicksort [y|y<-xs,y<x]
  ++[x]
  ++ quicksort [y|y<-xs,y>=x]
```