Programming Skills: Functional Programming

Haskell
A language for functional programming
Purpose

- General-purpose, functional
- Strongly typed, type inference
- Interpreted or compiled

- Documentation, downloads, tutorials
  http://haskell.org/

- Talk by Rhys Price-Jones and Jacob Rigby
Use

- **ghc, ghci compiler and interpreter**
  

- **hugs interpreter**
  

```bash
$ ghci
Prelude> "Hello World!"
"Hello World!"
Prelude> :quit

$ hugs
Hugs.Base> "Hello World!"
"Hello World!"
```
The Qualities of Haskell

• Pure functional
  – no side effects
  – value returned by a function depends on input and nothing else
  – intuitive
  – familiar (c/f spreadsheets, SQL)
The Qualities of Haskell

• Abstraction
  – Resource allocation is automatic
  – Sequencing is invisible
  – By definition, imperative languages cannot achieve this level of abstraction

• Intuitive
  – smaller semantic gap between algorithmic idea and implementation in a language
The Qualities of Haskell

• Laziness
  – nothing is evaluated until it is needed

• Strong typing
  – with automatic type inference
  – and usable polymorphism

• Elegance of expression
  – and conciseness
Execution Model

- **expression evaluation:**
  "Hello World!"
  ...

- **definition:**
  
  let header = value ;... -- ghci only

  :load file.hs
  header = value
  ...

12/19/06 haskell -7
Lexical Aspects

- **Input Format**: line-oriented, layout matters (can use braces and semicolons instead)
- **Comments**: -- to end of line, `{-- ... --}` multi-line, can be nested
- **Delimiters**: operators, white space
- **Reserved Words**: let where etc.
- **Keywords**: no
- **Names**: alphanumerics plus ‘_’, case-sensitive, different namespaces: `var Ctor`
Literate Programming

- file.lhs is assumed to contain comments and code must be marked explicitly:

  ```haskell
  repeat' digit
  returns an infinite list of the same digit
  > repeat' n = n : repeat' n
  ```

- Instead, for \LaTeX, code can be delimited by \begin{code} and \end{code}.
Lazy Evaluation

- Operands are bound to parameters
- Arguments are computed only as needed

```haskell
add as bs
returns a list of the element-wise sum of two infinite lists

> add (a:as) (b:bs) = (a+b) : add as bs

ints
returns a list of all integers

> ints = 1 : add ints (repeat' 1)
```
Modules

module Name (name, ...) where

- defines a module Name and optionally externally visible names.

import qualified Name hiding (name, ..)

- imports a module Name, optionally requests that all uses must be qualified, and optionally limits or hides imported names.

> module Functions where
> import Prelude hiding (take)
Patterns and Clauses

- Function parameters can be patterns
- Function definitions can consist of several alternative clauses grouped together
- Clauses are tried sequentially; one must succeed

```
take number-of-elements list
returns the first number-of-elements (or fewer) elements from the list

> take 0 xs = []
> take n (x:xs) = x : take (n-1) xs
```
Guards are tried sequentially; one must be True (or otherwise)

take number-of-elements list
returns the first number-of-elements (or fewer) elements from the list

> take n (x:xs)
>   | n <= 0   = []
>   | True     = x : take (n-1) xs
where { }

evens
returns a list of all even numbers

> evens = add ints ints

odds
returns a list of all odd numbers

> odds = sub evens (repeat' 1)
>   where
>     sub (a:as) (b:bs) = (a-b) : sub as bs

- establishes a local name space
- enclosed by braces or indentation
- can be nested

http://haskell.org/onlinereport/lexemes.html#sect2.7
let { } in

fibs
returns a list of all Fibonacci numbers

> fibs =
>   let
>     f a b = a : f b (a+b)
>   in
>     f 0 1

- same effect as where:
  - establishes a local name space
  - enclosed by braces or indentation
  - can be nested

http://haskell.org/onlinereport/lexemes.html#sect2.7
Tail Recursion

fac number
returns the factorial of a number

> fac n = f n 1
>   where
>   f 0 result = result
>   f n result = f (n-1) (n*result)

if the recursive call is the last operation it might be optimized as a loop.
tail recursion might be accomplished by passing the eventual result as a parameter.
Ackermann’s Function

\[ \text{ack} \ x \ y \]
returns the value of Ackermann's function
-- only \( \text{ack} \ 0 \ 0 \) through \( \text{ack} \ 3 \ 4 \) are reasonable

> \( \text{ack} \ 0 \ y = y + 1 \)
> \( \text{ack} \ x \ 0 = \text{ack} \ (x-1) \ 1 \)
> \( \text{ack} \ x \ y = \text{ack} \ (x-1) \ (\text{ack} \ x \ (y-1)) \)

- values grow very fast.
- usually used as compiler benchmark.
All Rationals

rats returns a list of all rational numbers

> rats = (1,1) : map next rats
>   where
>     next (n,1) = (1, (n+1))
>     next (n,m) = ((n+1), (m-1))

traverses a grid indexed by row and column diagonally to produce pairs of coordinates.

(x, ...) is a (heterogeneous) fixed-length tuple.

[x, ...] is a (homogeneous) variable-length list.

map \( f \) \( list \) applies a function to each element of a (single) list.
List Comprehension

primes
returns the list of all primes using the Sieve of Eratosthenes

```haskell
> primes = sieve [2..]
>   where
>     sieve (p:x) = p :
>                   sieve [n | n <- x, n `mod` p > 0]
```

- `[a,b..]` generates an arithmetic progression.
- `[expression | generator, filter, ...]` generates a list.
  - `pattern <- expression` is a generator.
  - `filter` is any boolean expression.

**case of** { } { }

```haskell
qsort list
returns sorted list

> qsort list =
>   case list of
>     [] -> []
>     (pivot:xs) -> qsort [x | x <- xs, x <= pivot]
>                   ++ [pivot] ++
>                   qsort [x | x <- xs, x > pivot]
```

case expression of

- permits selection by pattern matching
- enclosed by braces or indentation

http://cs.anu.edu.au/Student/comp1100/haskell/tourofsyntax.html#Patterns
http://haskell.org/onlinereport/lexemes.html#sect2.7
if then else

perms list
returns all permutations of a list

> perms [] = [[]]
> perms s =
>   [x:y | x <- s, y <- perms (s `without` x)]
>   where
>     without [] _ = []
>     without (x:xs) y =
>       if x == y then xs
>       else x : (xs `without` y)

- if allows selection of one of two values of the same type.
- guards allow many conditions.
- _ is an unbound parameter.
hanoi n
solves the Tower of Hanoi problem:
list of pairs indicating piles to move from and to

> hanoi n = f n 1 2 3
> where
>    f 0 _ _ _ = []
>    f (n+1) from help to = f n from to help ++
>        [(from, to)] ++
>        f n help from to
n Queens

```haskell
queens n
places n queens on an n by n board:
lists containing per column the row of the queen

> queens n = place n
>  where -- collect 1..n which do not conflict
>    place 0 = [[]]
>    place m = [ row:rest | rest <- place (m-1),
>                   row <- [1..n], safe row rest 1 ]
>    where -- check if row or diagonal in use
>      safe row [] _        = True
>      safe row (r:rs) i
>        | row == r         = False
>        | abs(row-r) == i  = False
>        | True             = safe row rs (i+1)
```
All Perfect Numbers

\[ \text{> perfects} = \]
\[ \text{filter } (\ n \rightarrow \text{sum (factors n)} == n) \text{ ints} \]

\text{factors n}
returns the list of factors of a number

\[ \text{> factors n} = \]
\[ \text{[i} \mid \text{i }\leftarrow [1..n \ `\text{div}` \ 2], \text{n }\ `\text{mod}` \ i == 0] \]

- \text{div and mod produce Integer results.}
- `'name` allows infix use.
- `\ pattern ... -> expression` creates an anonymous function.
- filter returns list elements matching a predicate.
Currying

perms' list
returns all permutations of a list

> perms' [] = [[]]
> perms' s =
>    [ x:y | x <- s, y <- perms' (filter (x /=) s)]

■ /= denotes inequality.
■ if a function call has too few arguments it returns a function.
Composition

pascal
returns a list of all rows of Pascal's triangle

> pascal = f [1]
>   where -- collect rows
>     f row = row : (f.next) (0:row)
>       where
>         next (x:y:tail) = (x+y) : next (y:tail)
>         next [x] = [x]