Introduction to Haskell

Associated code files to read, execute in Hugs, modify:
Intro.lhs, Trans1.hs, IntroFuns.lhs, ListOps.hs, Tuples.lhs

Functional Programming

- Haskell is a functional programming language:
  - Persistent data values and data structures (once built are never changed).
  - Recursion is a primary control structure.
  - Heavy use of higher-order functions (i.e., functions which take functions as data and return functions as results).
  - Functional composition as main means of composing programs using parts.
  - Example languages: Scheme, Miranda, ML, Ocaml.

- Imperative languages, by contrast, emphasize:
  - Mutable variables and data structures.
  - Looping rather than recursion is emphasized.
  - First-order rather than higher-order functions.
  - Examples: C, FORTRAN, Pascal, Ada, C++, Java, Perl.

Computing with expressions

- Haskell is an expression language. A program is an expression. The "meaning" of the program is the value of the expression.

<table>
<thead>
<tr>
<th>Prog1</th>
<th>Prog2</th>
<th>Prog3</th>
<th>Prog4</th>
<th>Prog5</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 + 25</td>
<td>square (3 + 5)</td>
<td>sum [1..10]</td>
<td>square (gcd (243, 16))</td>
<td>inRange (sum (fifthFact 6))</td>
</tr>
</tbody>
</table>

Haskell interpreter: Hugs 98

- Provides an almost complete implementation of Haskell 98.
- The environment to be used in this tutorial.
- Hugs is implemented as an interpreter.
- Used on average different machines from PCs to Unix boxes.
- A read-eval-print loop for displaying the value of each expression that is entered into the interpreter.
- Fast loading, type checking, and compilation of Haskell programs, with facilities from automated building of imported modules.
- Haskell is a typed language, that uses type inference as aid to the programmer, who then doesn't have to type every expression.
- Writing a Haskell program consists of giving a set of definitions for named data values, named data types, and named functions.
- A Haskell program uses these definitions in the formation of expressions. These expressions use the definitions to reduce the expression to eventually a value.
- Repeat: Haskell deals only with expressions. There are no statements to be used only for side effects.
### Giving things names

```haskell
main> inchesPerMile = 12*3*1760
main> drivingAge = 17
```

### Primitive types

- Int, Integer, Float, Double
- Bool values: True, False
- basic ops: &&, || :: Bool -> Bool -> Bool
- not :: Bool -> Bool
- Char with basic ops: ord :: Char -> Int, chr :: Int -> Char

### Simple named functions

- `square` :: Int -> Int
- `cube` :: Int -> Int
- `fac` :: Int -> Int
- `fact'` :: Int -> Int
- `newFac` :: Int -> Int

```haskell
square :: Int -> Int
square x = x * x

cube :: Int -> Int
cube x = if x == 0 then 1 else x * cube (x - 1)

fac :: Int -> Int
fac n = if n == 0 then 1 else n * fac (n - 1)

fact' :: Int -> Int
fact' n = if n == 0 then 1 else n * fact' (n - 1)

newFac :: Int -> Int
newFac n = if n == 0 then 1 else newFac (n - 1)
```

### Built-in composite types

- Lists
  - The working data structure home of functional programming.
  - For storing a collection of homogenous data values.
  - We start with some literal list values:
    ```haskell```
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
    ```haskell```
**Built-in composite types**

**Constructing Lists**

- Haskell provides numerous built-in operations that manipulate and return lists.
- The most basic one creates a new list by adding an element to the front of an existing list.
- The operation is written: and pronounced cons
  - `a`::{"b","c","d"} evaluates to `a`::{"b","c","d"}
  - "hello"::{"there"} evaluates to "hello", "there"
- Other list values:
  - `[1..10]` evaluates to `[1,2,3,4,5,6,7,8,9,10]
  - `[3,5,7]` evaluates to `[3,5,7]

**List operations**

**Pattern matching**

- Pattern matching is generally used to access specific values of data structures among others. It's commonly used in the following two cases:
  - function definition.
  - Initialization of available.
- You specify a pattern which specifies a value or value decomposition for data structures, and during run-time if the pattern is matched the instructions associated to it are executed.

**Built-in composite types**

**List operations**

- Haskell provides numerous operations on lists. Some examples:

  ```haskell
  head   :: [a] -> a
  head (x:_@) = x
  last   :: [a] -> a
  last [x]  = x
  last (_:xs) = last xs
  tail   :: [a] -> [a]
  tail (_:xs) = xs
  length :: [a] -> Int
  length [] = 0
  length (x:xs) = 1 + length(xs)
  (++)    :: [a] -> [a] -> [a]
  []     ++ ys         = ys
  (x:xs) ++ ys      =  x : (xs ++ ys)
  filter :: (a -> Bool) -> [a] -> [a]
  filter p []       =  []
  filter p (x:xs) = if p x
                       then x : (filter p xs)
                       else filter p xs
  map    :: (a -> b) -> [a] -> [b]
  map f []         =  []
  map f (x:xs )  =  f x : (map f xs)
  replicate :: Int -> a -> [a]
  replicate n x     = take n (repeat x)
  ```

See Thompson's pg 91.
Prog  add the elements of an [Int]

\[
\text{sum}0\ \text{list} = \begin{cases} 
0 & \text{if } (\text{length}\ \text{list}) = 0 \\
\text{else} & (\text{head}\ \text{list}) + \text{sum}0\ (\text{tail}\ \text{list}) 
\end{cases}
\]

\[
\text{sum}\ \text{list} = \begin{cases} 
0 & \text{if } (\text{length}\ \text{list}) = 0 \\
\text{otherwise} & (\text{head}\ \text{list}) + \text{sum}'\ (\text{tail}\ \text{list}) 
\end{cases}
\]

\[
\text{sum}'\ \text{list} = \begin{cases} 
0 & \text{if } (\text{length}\ \text{list}) = 0 \\
\text{otherwise} & (\text{head}\ \text{list}) + \text{sum}'\ (\text{tail}\ \text{list}) 
\end{cases}
\]

\[
\text{sum}''\ [x] = x
\]

\[
\text{sum}''\ (x:xs) = x + \text{sum}''\ xs
\]

Tuple pattern matching

- Pattern matching is applicable for tuples. In particular to define functions on tuples you use a pattern
- Example:
  - \(\text{flip} :: (\text{Int}, \text{Int}) \to (\text{Int}, \text{Int})\)
  - \(\text{flip}\ \text{pair} = (\text{snd}\ \text{pair}, \text{fst}\ \text{pair})\) -- better below:
  - \(\text{flip}\ (x,y) = (y, x)\) -- match

- \(\text{equal} :: (\text{Int}, \text{Int}) \to (\text{Int}, \text{Int}) \to \text{Bool}\)
- \(\text{equal}\ (x0,y0)\ (x1,y1) = x0 \times y1 == x1 \times y0\)

- \(\text{numberChildren} :: [\text{String}, \text{Bool}, [\text{String}]] \to \text{Int}\)
- \(\text{numberChildren}\ \text{name, married, children} = \text{length}\ \text{children}\)

Built-in composite types: Tuples

- Used for the grouping of values of possibly different types.
- Haskell: structs (in C), records (in Ada, Pascal)
- Examples:
  - \([0, 3] :: (\text{Int}, \text{Int})\)
  - \(\text{"Mary", "Joseph"} :: (\text{String}, \text{Bool})\)
  - Tuples with two components are common, so Haskell provides two functions: \(\text{fst}\) and \(\text{snd}\) which give the first and second component respectively of a given pair.

- Important to distinguish between tuples and Haskell lists
  - A tuple has a fixed number of possibly heterogenous values, and where each entry is of a predetermined type.
  - A list has a non-pre-determined number of homogenous values.

Tuple uses

- They play the same role that structs in C or records in Pascal or Ada play, the only difference with those is that by default Haskell values do not require labels.
- They can be used for the definition of user-defined types.
- They can also be used to return more than one value from functions.

- \(\text{type}\ \text{Date} = (\text{Int}, \text{Int}, \text{Int})\) -- (day, month, year)
- \(\text{type}\ \text{TestGrade} = (\text{String}, \text{Int}, \text{Char})\) -- (id, score, grade)
- \(\text{type}\ \text{Employee} = (\text{Date}, \text{String}, \text{Float})\) -- (begin, name, salary)

- \(\text{type}\ \text{Author} = \text{String}\)
- \(\text{type}\ \text{AuthorPub} = (\text{Author}, \text{Int})\) -- (author, book pubCount)

- \(\text{pub\Count} : \text{AuthorPub} \to \text{Author} \to \text{AuthorPub}\)
- \(\text{pub\Count}\ (\text{author}, \text{pubCount}) = (\text{author}, \text{pubCount} + 1)\)
- \(\text{if}\ author =\ sobre\ then\ \text{author} =\ author\)
Defining lists: List comprehension

- Characteristically found in functional programming languages.
- A list comprehension is used to define the elements of new list in terms of elements of another list.
- The list being used is called the generator.

evenNumbers = [2*n | n <- [1..4]]  \[2,4,6,8\]
shiftValues = [n + 1 | n <- [5,8,11,15]]  \[6,9,12,16\]
primes = [ isPrime n | n <- [2..20]]  \[2,3,5,7,11,13,17\]

We can combine generator with a predicate:
shiftValues' = [n + 1 | n <- [5,8,11,15], isEven n]  \[6,9\]
digitList = [ ch | ch <- string, isDigit ch]

- We can use patterns for the list generator values to use:
  palindrome = [ n + m | (n,m) <- palindrom]
  palindrome' = [ n + m | (n,m) <- palindrom, m mod n == 0]

Prelude library

- This is a module that contains definitions for built-in types and functions over these types.
- This library is loaded automatically when invoking hugs.
- It is referred to as the standard Prelude.
- As it has grown it has been divided into several other standard modules.
- When you program, if you happen to define a name already defined in the Prelude, you will get an error.
- Example: defining String to be type String = [Char]

**ERROR:** "c:\unocourses\4501\haskell\code\SimpleFuns.hs":27 - Ambiguous type constructor occurrence "String"
*** Could refer to: Main.String Prelude.String
Warning: error messages in Haskell are only partially helpful.

- Read the first 8 chapters of Thompson's book.
- He goes slowly and easy
- You must not only read but program
- Read as well the programs available on line.
- Exercises:
  4.1 - 4.6, 4.9, 4.10, 5.3, 5.9, 5.10, 511, 7.2, 7.3, 7.7, 7.8, 7.14, 7.16, 7.17, 7.18, 7.26