Haskell subroutines and parameter passing, exception handling, and data abstraction

By Caroline Cox, Trang Dang, and Vy Pham

Subroutines and Parameters Passing

Subroutines

- Haskell is a functional language, thus it only uses functions and not procedures:
    - Subroutines cannot have side effects, meaning that various internal states of the program will not change. Functions will always return the same result if repeatedly called with the same arguments.
    - Thus, Haskell only supports functions, since procedures that do not return a value have no use unless they can cause a side effect.

- Example of function in Haskell: Haskell has Type Inference, so stating types in function declaration is not needed, but still encouraged for clarity

```
add :: Integer -> Integer -> Integer    --function declaration
add x y =  x + y                        --function definition

main = do
   putStrLn "The addition of the two numbers is:"
   print(add 2 5)     --calling a function
```

- Pattern Matching: Pattern Matching helps simplify code by matching specific type of expressions.
    - Function that says the numbers from 1 to 5 and says "Not between 1 and 5" for any other number
    - The last line is a catch-all which catches any pattern that does not match 1 to 5

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

    - Haskell does not have a "for" loop, so recursion is an integral part of Haskell programming

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact ( n - 1 )

main = do
   putStrLn "The factorial of 5 is:"
   print (fact 5)
```

- The compiler starts by searching for function "fact" with an argument.
- If the argument is not equal to 0, then the number will keep on calling the same function with 1 less than that of the actual argument.
- When the pattern of the argument exactly matches with 0, it will call our pattern which is "fact 0 = 1"

Parameters Passing

- Haskell uses lazy evaluation
  - Expressions are not evaluated when they are bound to variables, but their evaluation is deferred until their results are needed by other computations.
  - Arguments are not evaluated before they are passed to a function, but only when their values are actually used

- Thus, Haskell uses call-by-name, where arguments are substituted directly into the function body and then left to be evaluated whenever they appear in the function.

- Call by name:
  - On invocation:
    - Substitute textually the names (expressions) of actual parameters into text of function
    - Execute the function
  - On return:
    - Pop the stack frame

Data Abstraction

- As a functional programming language, Haskell has more mechanisms for abstraction than an imperative language like C++ or Java
  - Haskell programs focus on what is being computed as opposed to how to compute it. The implementation of how to compute something is hidden by a computational interface that acts as a function
  - Automatic garbage collection, although not directly used by the user, eliminates the need for freeing up memory in bits, acting as an abstraction to distance the user from the hardware

- Haskell doesn't have OOP because you can't group data and functions into a single "object"
- Inheritance can be emulated in Haskell but it's not really necessary
    - As long as the data type fulfills the requirements of a class it can be instantiated as an instance of that class and doesn't need to be a child class (no need for inheritance)
- Haskell has two mechanisms for using ADTs, classes and modules
    - Implementation of data type is written in a separate module and only the interface is exported/visible to the user
    - Both mechanisms must be used together in order to encapsulate type classes
        - Module export list:

```
module Stack (Stack, empty, push, pop, top, isEmpty) where

newtype Stack a = Stk [a]

empty                  = Stk []
push   x (Stk xs)      = Stk (x:xs)
pop      (Stk (x:xs)) = Stk xs
top      (Stk (x:xs)) = x
isEmpty  (Stk xs)      = null xs
```

            - Constructor is hidden (if it weren't, method signature would include Stack(Stk))
            - Stack can only be created using public facing methods empty, push, and pop and examined with top and isEmpty
    - Aspects of both mechanisms
        - Internal implementation hidden from user (encapsulation): hidden constructors and no pattern matching
        - Data is accessed through methods or operators, which are exposed in a signature
        - For example, one method of a Stack ADT is push, whose signature would look like this: `push :: a -> Stack a -> Stack a`
- Built-in and user-defined ADTs are possible
    - Examples of built-in ADTs are the primitive types Integer and Float
    - Example of user-defined ADT is the Stack type mentioned above
    - Haskell comes with the Show type class, of which a type can be instantiated by providing the type with a show function that converts the type to a string
- Parametrized data type can also be thought of as ADTs because some information in the type can be left undefined
    - Example:

```haskell
data Tree a = Nil
            | Node { left  :: Tree a,
                     value :: a,
                     right :: Tree a }
```

- Example of elements being defined in parametrized data type above:
- ```haskell
  three_number_tree :: Tree Integer
  three_number_tree = Node (Node Nil 1 Nil) 2 (Node Nil 3 Nil)
  ```

Exception Handling

- Distinction between “exception” and “error” in the articles from https://wiki.haskell.org/
  - Exceptions - Expected but irregular situation at runtime
  - Errors - Mistakes that can only be resolved by fixing the program

⇨ So an unhandled exception could be considered an error

- Four standard ways to handle exceptions and errors

Exception: basic functions handling exceptions from Control.Exception

- `throw :: Exception e => e -> a`
- `try :: Exception e => IO a -> IO (Either e a)`

- Use of `throw`:

```haskell
data MyException = MyException
  deriving (Show, Typeable)
instance Exception MyException

main :: IO ()
main = runSimpleApp $ do
  logInfo "This will be called"
  throwIO MyException
  logInfo "This will never be called"
```

*show: class, types that can be converted to String
*typeable: associate representation to types

- Use of `try`:
```

```
ghci> :m Control.Exception
ghci> let x = 5 `div` 0
ghci> let y = 5 `div` 1
ghci> print x
*** Exception: divide by zero
ghci> print y
5
ghci> try (print x)
Left divide by zero
ghci> try (print y)
5
```

Error wouldn't be thrown at `let x = 5 'div' 0` because of lazy evaluation: `x = 5 'div' 0` isn't evaluated until we use it. Therefore, we could use the function `evaluate` to force early evaluation of x.

```
ghci> result <- try (evaluate x)
Left divide by zero
```

Errors:
When some function contains error, causing the evaluation to crash.

```
head :: [a] -> a
head (x:_) = x
head []    = error "empty list"
```

Here, `head` is taking in a list and return the first element in that list, and there would be an error when `head` is used on an empty list.

`error` is a function that represents an error and a message.

We can catch these errors with `evaluate` and `try` above!

Error using Maybe:

`Maybe` represents the possibility of an error. When an operation falls, we use the `Nothing` constructor, and when it doesn't, we use the `Just` constructor to wrap our values.

Example: return the "Name : [name of person]" when we can find the name, and "no name specified" when we can't.

```
case lookup name person of
  Nothing -> "no name specified"
  Just name -> "Name: " <> name
```

Error using Either

We would have two sides, `Left` and `Right` carrying a message. `Left` indicates an error and `Right` indicates success.

For example: if we have an error called `ParseError`, we added the type to the method signature.

```
runParser :: Parser a -> Text -> Either ParseError a
```

And then we define the left and right clauses.

```
main = do
  line <- getLine
  case runParser emailParser line of
    Right (user,domain) -> print ("The email is OK.",user,domain)
    Left (pos,err) -> putStrLn ("Parse error on " <> pos <> ": " <> err)
```

Sources:

https://wiki.haskell.org/Why_Haskell_matters
https://wiki.haskell.org/OOP_vs_type_classes
https://wiki.haskell.org/Abstract_data_type
http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/AbstractDataTypes.html
http://book.realworldhaskell.org/read/error-handling.html
http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html#deftypes.morecontrolled
https://www.fpcomplete.com/haskell/tutorial/exceptions/
https://wiki.haskell.org/Error