Report

Darrious Barger

About

This project was an extension of the imp programming language we did in a previous homework. My goal was to add support for multiple types, and support for functions. The program is run with `> ghci Main.hs` and then running the `Main` method. This will prompt the user for the name of a `.imp` file that contains code. I have included a number of test files in the test directory. Below are some details about the language syntax.

IfThenElse Statements

```
if (boolean) then { code block } else { code block }
```

While Loops

```
while (boolean) { code block }
```

Functions

```
def FuncName(arg1, arg2, argN) { code block followed by return }
```

Modules

The program is split into 6 modules: Parse, Lex, Test, Exec, Types, and Main. Each module provides functionality to the program. The Main module has a main method that runs various methods in the other modules;

Types

This module contains all the data types for our program including Env, AExpr, BExpr, FunDefn and so on. This module contains no functions

Lex

This module contains all logic for lexical analysis like our classify, preProcess, and lexer functions. The file input is turned into tokens.

Parse

This module contains logic for parsing. We take a list of tokens and reformat/classify them further into other tokens using our shift reduce function.

Exec

This contains logic for executing instructions. It contains methods such as lookup, evala, evalb, exec, and helper functions

Tests

This module contains various tests for different parts of the program.

Main

Contains the main method to run the program. The main function asks the user for the name of a file and sends the contents of that file to various functions in other modules.

```
main :: IO ()
main = do
  putStrLn "Enter a .imp file with code."
  filename <- getLine
  --let filename = "testFun.imp"
  contents <- readFile filename

  let lexed = lexer contents
  putStrLn "Here is the result of lexical analysis:"
  putStrLn (show lexed)
  putStrLn "-------------------------------------------"

  let parsed = sr [] $ lexed
  putStrLn "Here is the result of parsing:"
  putStrLn (show parsed)
  putStrLn "------------------------------------------"

  let update = updateDefs parsed
  let removed = removeDefs parsed
  let parse = exec update (Do(reverse $ listInstr removed)) []
  putStrLn "Here are the functions:"
  putStrLn (show update)
  putStrLn ""
  putStrLn (show  (removed))
  putStrLn ""
  putStrLn "Here is the result of the program:"
  putStrLn (show parse)
```

The function reads a file as a string and passes it to our lexer. Those results get passed to sr (shift reduce). Those parsed results get sent to the exec function to get the results of the program. The updateDefs function is a helper that searches for all function definitions and stores in a list of definitions. The list of definitions is searched when a function is called.

Difficult Features

Making the program support multiple types of variables was difficult because it required changes to many functions. I will provide an example below in the `lookup` method.

```
lookUp :: Vars -> Env -> Values
lookUp x [] = VNull
lookUp x ((a, IntPrim num) : b)   = if (a == x) then IntPrim num else
(lookUp x b)
lookUp x ((a, BoolPrim bool) : b) = if (a == x) then BoolPrim bool
else (lookUp x b)
```

The `Values` data type consists of prims such as `IntPrim` and `BoolPrim`. Before, the program only supported integer variables, but now, we must pattern match specific types when looking up a variable in an environment. Though this change seems simple enough in `lookup`, this feature requires a number of design changes in the programming. The language also allows you to use a variable as a condition in an if or while loop.   There is very minor error checking as seen below.

```
extractInt :: Values -> Vars -> Integer
extractInt (IntPrim x) _ = x
extractInt x var = error ("Actual value (" ++ show x ++
                          ") - Expected IntPrim -  Make sure var "
                     ++ (show var) ++ " has been initialized and is
correct type")
```

Supporting functions was a difficult feature as well. It required the parsing of functions including the name, list of arguments, and instruction block. Once parsed, the program will be able to execute the function when called. This is done by making changes to `sr` and the `evala` function. Some helper functions and new data types had to also be created such as

```
type FName = String
data FunDefn = Function FName [Vars] [Instr]  deriving Show
type Defs = [FunDefn]
```

Once the file is parsed, I made a helper method to filter through the program and generated `Defs` based on function definitions. This is important because `Defs` is required for all of the evaluation functions when we execute a function call. Here is the method.

```
updateDefs :: [Token] -> Defs
updateDefs [] = []
updateDefs (PI (Do a) : PA (FApply name expr) : Keyword "def" : ts)
          = [Function name (getVar expr) a] ++ (updateDefs ts)
updateDefs (a:ts) = updateDefs ts
```

The functions in this programming language support recursion. They do support defining a return type yet (any type can be returned, but the program does not check if you are assigning the correct type to the returned value). To see the program executing functions, see the example files.

Future Features

There are a lot of cool features that could be added. For one, more robust error checking on the types when assigning variables to something. The ability to add specific return types to a function could also be added. Other features include an object or struct system, supporting various data structures like linked list, arrays, etc.

Summary

This program parses and executes a basic programming language. The language supports Bool and Integer types and also supports functions (including recursion). The language also fully supports if statements and while loops (and variables can now be used as conditionals in these). I have learned a lot from this project, not only about Haskell, but about program language design. I want to continue to refine this project and become more proficient with Haskell.