Tutorial 5

The goal of this tutorial is to extend the language to allow top-level functions. We will also augment the type checker for type checking top-level functions.

Tutorial Bundle

For this tutorial there is a bundle of files that can be downloaded from Moodle. Look for Bundle for Tutorial 5 under section *Tutorials and Assignments*.

Top-level Functions

So far we have been fiddling around a small language that supports arithmetic and logic operations, variable declarations and so on. One big feature in almost all modern programming languages is the ability to define *functions*. Generally speaking, A function is a program fragment that "knows" how to accomplish a specific task. For example, in C language, you can define a function that computes the average of an array of numbers. Once that function is defined, you can use it anywhere for any times without having to rewrite it. Functions provide an abstraction layer, that is, it "encapsulates" many instructions into a single piece of code, thus increasing the readability and code reuse of programs.

A function definition consists of a function name, a list of parameters and a body. For the matter of implementation, the syntax of function definitions in our language is the following:

$$\textbf{function}\ \mathsf{fname}\,(\mathsf{parameter}_1 : \mathsf{type}_1, \ldots, \mathsf{parameter}_n : \mathsf{type}_n)\,\{\,\mathsf{body}\,\}$$

each parameter name must be accompanied with a type (`Int` or `Bool` for this tutorial).

The syntax of function calls is the following:

$$\mathsf{fname}\,(\mathsf{expression}_1, \ldots, \mathsf{expression}_n)$$

Since programs written in our language so far are composed of expressions only, to allow for functions, we will add another form of program components, which will hold all the function definitions. In other words, a complete program now will look like:

$$\mathsf{function}_1\ \mathsf{function}_2 \ldots \mathsf{expr}$$

Note that you can have multiple (or none) function definitions, and they must be placed together. But you can have only **one** expression, and it must come last.

Here is an example of a complete program:

```
function absolute(x : Int) { if (x > 0) x; else -x }
function max(x : Int, y : Int) { if (x > y) x; else y }
max(absolute(-5), 4) // return 5
```

Implementing Top-level Functions

In order to allow for function definitions, we define a new datatype `Program` as follows:

```
data Program = Program FunEnv Exp
```

Each program has a *function environment*, which holds all the function definitions:

```
type FunEnv = [(String, Function)]
```

The above definition is read as: a function environment consists of pairs of function names and their associated bodies.

A function body is then defined as:

```
data Function = Function [(String, Type)] Exp
```

The list type following `Function` constructor denotes parameters, each of which is a pair of parameter name and its type.

To be able to call functions, we need one extension to the expression datatype:

```
data Exp = ...
         | Call String [Exp]
```

Here is the whole abstract syntax for the exmaple program as shown in the last section:

```
Program [("absolute"
         ,Function [("x",TInt)]
                   (If (Bin GT
                            (Var "x")
                            (Lit (IntV 0)))
                       (Var "x")
                       (Unary Neg (Var "x"))))
        ,("max"
         ,Function [("x",TInt),("y",TInt)]
```

```
                (If (Bin GT
                         (Var "x")
                         (Var "y"))
                    (Var "x")
                    (Var "y")))]
        (Call "max" [Call "absolute" [Lit (IntV (-5))],Lit (IntV 4)])
```

Pretty scary, right? That's why we need a new pretty printer!

Pretty Printer, revised

The new pretty printer should be able to handle function definitions. But don't worry, I have completed most parts of the function, and your job is to render function definitions nicely.

A little detour first. Let's define a function `showSep sep p` that takes an element and a list of strings, and "intersperse" that element between the strings of the list. A few examples below:

```
*Declare> showSep "; " ["hello", "to", "world"]
"hello; to; world"

*Declare> showSep "; " ["hello"]
"hello"

*Declare> showSep "; " []
""
```

Notice that when the second argument is a singleton list, it just prints out the string.

Question 1 (5pt): Define the function `showSep` in the file *Declare.hs*.

Let's then see how to print a function call expression. A function call consists of a function name, and a list of arguments. The type of each argument is `Exp`, so printing them is trivial (just apply `show` to each of them).

To render a function call, we first print the function name, followed by "(". After that, we apply `show` to the list of arguments, and get a list of strings. The list of strings should be separated by ", " (a comma followed by a space, remember to use `showSep`), and lastly closed by ")".

Question 2 (10pt): Complete the case for `Call f args` in the function `ShowExp`.

Here are some examples:

```
*Declare> show (Call "max" [(Lit (IntV 3)), (Lit (IntV 4))])
"max(3, 4)"

*Declare> show (Call "abs" [(Lit (IntV 3))])
"abs(3)"
```

Printing a function definition should be straightforward. We first print the keyword **function**, followed by the actual function name. Then we add "(" and print the list of parameters. To render parameters, for each pair, we print the parameter name, followed by " : ", and then the type, finally closed by ")". Printing the function body is trivial, just apply `show`, surrounded by "{}"

The above paragraph is lengthy and tedious, you will get a better picture by the following example:

```
("foo",Function [("x",TInt),("y",TInt)] (Bin Add (Var "x") (Var "y")))
```

would be rendered to

```
function foo(x : Int, y : Int) {
 x + y
}
```

Notice there is a **space** before "x" in the body.

Question 3 (10pt): Complete the definition of `showFun` in the *file Declare.hs*.

It's time to finish the whole pretty printer for the datatype `Program`. We are almost there actually! With the helper functions we have defined so far, it is easy to render the whole program. Remember that a program consists of a bunch of function definitions followed by an expression. For each function definition, we call `showFun` to render them, separated by "\n". For the last expression, call `show` and it's done!

Question 4 (10pt): Look for `instance Show Program where`, complete the definition.

When you finish, you can try it out by rendering the example program `prog1` provided in the file *Declare.hs*. Your output should be *identical* to the following:

```
function absolute(x : Int) {
 if (x > 0) x; else -x
}
function max(x : Int, y : Int) {
```

```
 if (x > y) x; else y
}
max(absolute(-5), 4)
```

Evaluating Programs

As before, we need to again extend the evaluator with the case for function calls. We now use a top-level execution function:

```
execute :: Program -> Value
execute (Program funEnv main) = evaluate main [] funEnv
```

The new evaluator takes one additional argument: the function environment

```
evaluate :: Exp -> Env -> FunEnv -> Value
```

In the case for function calls, we first need to extract the function definition we are going to call. This can be achieved by "looking up" the callee function in the function environment, which in return will give us the parameter names and the function body (if the function has been defined before). After that, we perform evaluation on each argument and get a list of values. We then zip the parameter name with the corresponding argument value to form a new environment. Finally we evaluate the body expression in the new environment.

Though this sounds complicated, it's actually the standard mechanism of function calls. Besides, recursive functions now come for free! Behold:

```
function fact(x : Int) {
 if (x == 0) 1; else x * fact(x - 1)
}
fact(5)
```

would evaluate to 120!

Question 5 (15pt): Complete the case for function calls in the function `evaluate` in the file *Interp.hs*.

The Glorious Type Checker

The glorious type checker prevents us doing something crazy, e.g., calling functions that are not even existent, or passing the wrong arguments that are incompatible with the formal parameters.

To augment the type checker to allow type checking top-level functions, we need two steps: 1) type checking function definitions; 2) type checking function calls.

Let's first see how to type check function definitions. In order to do that, we will have to define another data structure called *function type environment* to associate function names and their corresponding parameters and return types:

```
type TFunEnv = [(String, (TEnv, Type))]
```

that is, each pair consists of a function name, another pair of parameters (names and types) and the function return type. After successfully type checking function definitions, we will get a data structure of type `TFunEnv`, which is needed to type check function calls. So the type checker now will take one additional argument:

```
tcheck :: Exp -> TEnv -> TFunEnv -> Maybe Type
```

The function `checkFunEnv` is used to type check function definitions:

```
checkFunEnv :: FunEnv -> Maybe TFunEnv
checkFunEnv fds = checkFunEnv1 fds [] -- starts with an empty function type environment
```

inside `checkFunEnv1`, we type check functions by traversing through `fds`. For each function, we type check (by using `tcheck` as before) the function body in the old environments (i.e., the parameters and the function type environment). If the function body type checks, we will get a return type for that function, then we extend the function type environment by one element (the function name, the parameters and the return type).

Note that if any one of the function bodies fails to type check, `checkFunEnv` returns `Nothing`.

Question 6 (20pt): In the file *TypeCheck.hs*, finish the definition of `checkFunEnv`.

Secondly, let's finish type checking for function calls. We first look up the function name that is to be called in the function type environment. If it is not found, we return `Nothing`; otherwise, we will get a pair of corresponding parameters and the return type of the callee function. Then we check if the types of the arguments match the types of the parameters. If it is not the case, we return `Nothing` as well; finally we type check the function body of the callee function.

Question 7 (20pt): Complete the case for `Call name args` in the function `tcheck`.

Now everything is ready to make the glorious type checker! We proceed by first type checking the function definitions (using `checkFunEnv`), if they type check, we continue by type checking the expression (using `tcheck`). If it type checks, we return `True`; otherwise we return `False`.

Question 8 (10pt): Finish the definition of `checkProgram` according to the explanation in the last paragraph.

Oh, SINH!

If you manage to read thus far, I'd like to propose a name for our new-born language, *SINH* (pronounced like "Sinch"). As you can guess (really?), *SINH* is a recursive acronym for *"SINH Is Not Haskell!"*. In this section, let's make SINH an executable that takes a file name from command line and runs the program in that file.

In the bundle, you will find a file named `Main.hs`, inside which there is a *main* function. You don't necessarily need to understand the details inside *main*. It essentially takes the command-line arguments, reads the file by the name of the first argument, type check and finally evaluate the program in the file.

To compile SINH, run the following command in the console:

```
> ghc Main.hs
```

GHC will compile the source file `Main.hs`, producing an *object* file `Main.o` and an *interface* file `Main.hi`, and then it will link the object file to the libraries that come with GHC to produce an executable called `Main` on Unix/Linux/Mac, or `Main.exe` on Windows.

Now you can create a file, say `demo.txt`, inside which you can write a program in SINH, for example:

```
function absolute(x : Int) { if (x > 0) x; else -x }
function max(x : Int, y : Int) { if (x > y) x; else y }
max(absolute(-5), 4)
```

then save and run the command as follows:

```
> ./Main demo.txt
5
```

Or if you use Windows:

```
> Main.exe demo.txt
5
```

Hurrah!

(Optional) Future Improvements

If you remember, SINH can run recursive programs happily. But now, if you try, the type checker will reject all recursive programs, why?

This is because when the type checker is checking the function body, it doesn't know yet the return type of the function being checked. Therefore, when you try to call the function itself inside the body, it thinks that you are calling a function that is not yet defined, thus rejecting the program.

However this can be improved (or corrected?) by requiring programmers to specify the return type of every function as in C and Java. The syntax of the function definition would then be like:

$$\textbf{function}\ \mathsf{fname}\,(\mathsf{parameter}_1 : \mathsf{type}_1, \ldots, \mathsf{parameter}_n : \mathsf{type}_n) : \mathsf{type}\,\{\,\mathsf{body}\,\}$$

In this optional question, you are encouraged to think about the necessary changes in datatypes that are required for such extension. You should also adapt/extend the parser, pretty printer, evaluator and type checker.

Again, this question is not graded, however, if you do it, it will give you bonus points, depending on how much further you improve SINH. (Be sure to document extra features you have implemented)

Assignment 4

Your forth assignment is to try to complete as many questions in the tutorial as you can. All questions (except for the optional one) are **graded**. You should submit your solutions to Moodle. Look for Submission of Tutorial 5 under section *Submissions and Score Reporting*.

The deadline for this assignment is **November 11, 11:55 PM**.

Please submit your answers using the modified Haskell files. **Try to ensure that the files you submit compile. If you submit files that do not compile, points will be removed from your grade. Try to comment the code explaining what you intended to achieve. If you cannot create code that compiles, it is especially important to write some comments explaining what you intended to achieve.**