Tutorial 3

In this tutorial we will review some of the concepts that we have learned so far in programming languages and add features to the simple interpreter we built last week and extend it to support local variables.

Tutorial Bundle

For this tutorial there is a bundle of files that can be downloaded from Moodle. Look for Bundle for Tutorial 3 under section *Tutorials and Assignments*.

A Language with Local Variable Declarations

The file *Declare.hs* contains the following definition for the abstract syntax of the language that we will be using:

```
data Exp = Num Int
         | Add Exp Exp
         | Sub Exp Exp
         | Mult Exp Exp
         | Div Exp Exp
         | Var String           -- new
         | Decl String Exp Exp  -- new
```

Compared to what we had last week, we add two new constructors `Var` and `Decl` to the datatype `Exp`, which allows our language to have local variables. For example, the expression

```
e1 :: Exp
e1 = Decl "x" (Num 3) (Mult (Var "x") (Num 3))
```

has the abstract syntax tree as shown in Figure 1.

Now you may be wondering what is the meaning of the `Decl` constructor? Or How can we interpret it? Take `e1` for example, the `Decl` constructor introduces a new variable named `x`, to which we assign value 3, and finally we multiply the variable `x` by 3. (Try to guess the result of this expression in your heart.)

Another example of our extended language would be

```
e2 :: Exp
e2 = Decl "x" (Add (Num 3) (Num 4)) (Var "x")
```

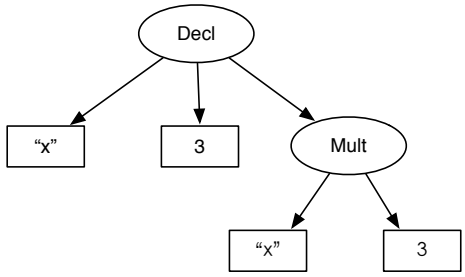

Figure 1: Abstract Syntax Tree of e1

In this tutorial, we assume that the concrete syntax for variable declarations is JavaScript-like, of the form:

```
var name = exp; exp
```

So `e2` is effectively equivalent to

```
var x = 3 + 4; x
```

Question 1: In the last tutorial, we knew how to build a pretty printer for our language. Now extend it to also support `Var` and `Decl` constructors. Here are some more examples for you to test your pretty printer:

```
e3 :: Exp
e3 = Add (Var "x") (Mult (Num 4) (Num 5))

e4 :: Exp
e4 = Decl "y" e3 (Div (Var "x") (Var "y"))
```

Here is the expected output in `ghci`:

```
*Declare> e3
(x + (4 * 5))

*Declare> e4
var y = (x + (4 * 5)); (x / y)
```

Variable Renaming

In Monday's lecture, we have seen how substitution works for an expression. Sometimes it is also useful to rename a variable in an expression. Here are some examples of renaming at work in `ghci`:

```
*Declare> rename "x" "i" e3
(i + (4 * 5))

*Declare> rename "x" "i" e4
var y = (i + (4 * 5)); (i / y)
```

Note that if a variable is bound within the expression, then renaming should not rename the variable inside that scope (similar to what happens in substitution). For example:

```
*Declare> rename "x" "i" e1
var x = 3; (x * 3)
```

Here the variable `x` is bound by `var x = 3`. So in the expression `x * 3`, renaming should not rename the variable `x`.

Question 2: Now you are asked to define the operation of renaming. Go to the file *Declare.hs* and look for the following definition:

```
rename :: String -> String -> Exp -> Exp
rename = error "TODO: Question 2"
```

The first argument of `rename` is the variable name to be renamed, and the second is the new name to replace with.

Interpreter, revised

Now it's time to extend our interpreter from last week to support local variables.

To do that, we need something called *enviroment* to help track all the variables and their associated values in an expression. Recall that in Monday's lecture, we have learned that a *binding* is an association of a variable with its value. We can represent *binding*s in Haskell as a pair. For example, `("x", 5)` in Haskell means that variable `x` is associated with value 5. We can use the `type` keyword in Haskell to introduce a type alias as follows:

```haskell
type Binding = (String, Int)
```

There can be multiple variables in a single expression. For example, our interpreter should be able to evaluate `2 * x + y` where $x = 3$ and $y = -2$. A collection of *binding*s is called an *environment*. In Haskell, we can represent *environment* as a list of pairs:

```haskell
type Env = [Binding]
```

Our extended interpreter now should work as follows:

1. It starts with an empty environment.
2. When encountering a number, it should just return the number as it is.
3. When encountering arithmetic operations, it recursively evaluates each operand, as we did last week.
4. When encountering a variable, it should look up the variable in the environment. If the variable is found, return the value associated with it; if not, signal an error.
5. The most interesting part is variable declarations. To evaluate that, we first evaluate the expression associated with the variable using the old environment, then we bind the variable with the value that results from the previous evaluation to form a binding, and augment the old environment with this binding, finally we evaluate the body of the variable declaration in the new environment.

Though the explanation is kind of daunting, it should be as straightforward as you might think.

Question 3: In the *Interp.hs* file, fill in the definition of `evaluate` as shown below.

```haskell
evaluate :: Exp -> Int
evaluate e = eval e [] -- starts with an empty environment
  where
    eval :: Exp -> Env -> Int
```

To test it out, we need to extend our parser to support local variables. Fortunately I have done that for you :) The bundle contains a modified Happy grammar file *Parser.y* from last week. Typing

```
> happy Parser.y
```

in the console will give you a new Haskell file *Parser.hs* as a drop-in replacement for the old parser.

Now you should be able to use our old friend `calc` to test your implementation of `evaluate`. For example:

```
*Interp> calc "var x = 3; x + 4"
7
```

Or if we have a undeclared variable:

```
*Interp> calc "var x = 3; x + y"
*** Exception: Not found
```

Of course the exception message could be more informative, like printing out the undeclared variable name for example.

Note that you should also test examples that have nested declarations. For example:

```
var a = 3; var b = 8; a + b   // evaluates to 11
var a = 3; var a = 4; a + 3   // evaluates to 7
```

That is to say, the inner bindings will shadow the outer ones if they happen to have the same name.

Multiple Bindings in Variable Declarations

Another extension we can add is to allow multiple bindings in a variable binding expression. For example:

```
var x = 3, y = 9; x * y
```

declares two variables at once.

In order to allow for this change, we will have to again modify the abstract syntax of `Exp`. The new language with multiple bindings can be expressed by adding a `DeclareMulti` constructor to support a list of pairs of strings and expressions:

```
data Exp = ...
         | DeclareMulti [(String, Exp)] Exp
```

Question 4: Modify the definition of `Exp` as stated above. You should also modify all associated functions: `showExpr`, `rename`.

Interpreter, revised again

Once again, our interpreter needs to be augmented. This time it should be almost identical to what we have for the case of `Decl`. However, if a `DeclareMulti` expression has duplicate identifiers, your program must signal an error. It is legal for a nested `Decl` to reuse the same name. Two examples:

```
var x = 3, x = x + 2; x * 2        // illegal
var x = 3; var x = x + 2; x * 2    // legal
```

The meaning of a `DeclareMulti` expression is that all of the expressions associated with variables are evaluated first, in the environment before the body is entered. Then all the variables are bound to the values that result from those evaluations. Then these bindings are added to the outer environment, creating a new environment that is used to evaluate the body. This means that the scope of all variables is the body of the `DeclareMulti` expression in which they are defined.

Note that a multiple declare is not the same as multiple nested declares. For example:

```
var a = 3; var b = 8; var a = b, b = a; a + b       // evaluates to 11
var a = 3; var b = 8; var a = b; var b = a; a + b   // evaluates to 16
```

Question 5: Modify the definition of `evalaute` to cover the case of `DeclareMulti`.

You can assume that the inputs are valid programs and that your program may raise arbitrary errors when given invalid inputs.

In the file *Interp.hs*, there are several test cases that you can use to test your implementation.

Also note that the parser in the bundle is unable to parse expressions with multiple bindings. If you are adventurous enough, you can go modify the Happy grammar file to add support for multiple bindings.

No Assignment

There is no assignment for this week, however, you should try to complete as many questions in the tutorial as you can. This will help deepen your understanding of how evaluation works in programming languages.

You can submit your answers to Moodle. Look for Submission of Tutorial 3 (optional) under section *Submissions and Score Reporting*, and I will provide feedback on your answers.