The University of Hong Kong

COMP3259: Principles of Programming Languages

Tutorial 3

Tutor
Wan Qianyong `qywan@cs.hku.hk`

Instructor
Bruno Oliveira `bruno@cs.hku.hk`

25 February 2025

Table of Contents

1 Introduction

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 time. We will extend the interpreter to support local variables.

2 A Language with Local Variable Declarations

First, please open `Declare.hs` under the `src` folder.

Recall that in the lecture, we were presented with the following definition for the abstract syntax of the arithmetic language we will be using (and later extending), similar to the following:

```haskell
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 time, we add two new constructors `Var` and `Decl` to the datatype `Exp`, which allows our language to have local variables. For example, the expression:

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

has the abstract syntax tree as shown below:

```
        Decl
       / |  \
      /  |   \
    "x"  3    Mult
              / \
             /   \
           "x"    3
```

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 then we multiply it by 3. (Ask yourself what is the result of this expression.)

Another example of our extended language would be:

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

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 and assignment, 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)
```

2.1 Variable Renaming

In the 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 $x \times 3$, 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:

```haskell
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.

3 Implementing Interpreter with Substitution

3.1 Multiple Bindings in Variable Declarations

You have learned in the lectures how to implement an interpreter based on the idea of substitution in order to support local 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 3. Modify the definition of `Exp` as stated above. You should also modify all associated functions: `showExp` and `rename`.

Question 4. Please implement a `subst` function that supports substitution of a binding on the extended language with `DeclareMulti`.

3.2 Interpreter, revised

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 in the outer environment before the body is evaluated. Then all the variables are simultaneously 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-variable declaration is not the same as multiple nested declarations. 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. Please implement an `evaluate` function based on the `subst` function.

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.

4 Implementing Interpreter with Environments

There is another way to implement our interpreter and support local variables. We need something called *environments* to help track all the variables and their associated values in an expression. Recall that in the lecture, we have learned that a *binding* is an association of a variable with its value. We can represent bindings in Haskell as a pair. For example, `("x", 5)` 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 $2 \times x + y$ where $x = 3$ and $y = -2$. A collection of bindings is called an environment. In Haskell, we can represent an environment as a list of pairs:

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

In terms of the operational semantics, we use the following grammar to describe environments:

$$\Delta ::= \cdot \mid \Delta, x = n$$

It reads that an environment Δ is either empty (represented by $\cdot$), or non-empty (contains at least one binding). The form of inference rules now becomes $\Delta \vdash e \to n$. 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.

$$\frac{}{\Delta \vdash n \to n}\ \textsc{En}$$

3. When encountering arithmetic operations, it recursively evaluates each operand, as we did last time. Note that since arithmetic operations don't change environments, we keep the same environment for both premises and conclusions.

$$\frac{\Delta \vdash e_1 \to n_1 \qquad \Delta \vdash e_2 \to n_2}{\Delta \vdash e_1 + e_2 \to n_1 + n_2}\ \text{E}+ \qquad \frac{\Delta \vdash e_1 \to n_1 \qquad \Delta \vdash e_2 \to n_2}{\Delta \vdash e_1 - e_2 \to n_1 - n_2}\ \text{E}-$$

$$\frac{\Delta \vdash e_1 \to n_1 \qquad \Delta \vdash e_2 \to n_2}{\Delta \vdash e_1 \times e_2 \to n_1 \times n_2}\ \text{E}\times \qquad \frac{\Delta \vdash e_1 \to n_1 \qquad \Delta \vdash e_2 \to n_2}{\Delta \vdash e_1 \div e_2 \to n_1 \div n_2}\ \text{E}\div$$

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.

$$\frac{\Delta(x) = n}{\Delta \vdash x \to n}\ \textsc{Evar}$$

Note that Δ(x) = n means "lookup x in Δ and bind the return value to n".

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.

$$\frac{\Delta \vdash e_1 \to n_1 \qquad \Delta, x = n_1 \vdash e_2 \to n_2}{\Delta \vdash (\textbf{var}\ x = e_1;\ e_2) \to n_2}\ \textsc{Edecl}$$

To get yourself familiar with the new inference rules, here is an example of the derivation tree for `e1`:

$$\frac{\cdot \vdash 3 \to 3 \qquad \dfrac{\dfrac{(x = 3)(x) = 3}{x = 3 \vdash x \to 3}\ \textsc{Evar} \qquad x = 3 \vdash 3 \to 3}{x = 3 \vdash x \times 3 \to 9}\ \text{E}\times}{\cdot \vdash (\textbf{var}\ x = 3;\ x \times 3) \to 9}\ \textsc{Edecl}$$

Question 6. Create a derivation tree for the expression:

```
var a = 3; var b = 8; a + b
```

You may create a text file `derivation.txt` where you can write the answer to this question.

Question 7. In the `Interp.hs` file, fill in the definition of `evaluate2` as shown below.

```
evaluate2 :: Exp -> Int
evaluate2 e = eval e []  -- starts with an empty environment
  where
  eval :: Exp -> Env -> Int
  eval = error "TODO: Question 4"
```

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`. You should be able to use our old friend `calc` to test your implementation of `evaluate2`. For example:

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

Or, if we have an unbound variable:

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

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

4.1 Nested Declarations and Variable Name Conflicts

The rules we defined above does not handle variable name conflicts for nested declarations. For example, deriving this expression:

```
var a = 3; var a = 4; a + 3
```

Our context Δ would contain two bindings for variable a, i.e. $a = 3$ and $a = 4$, which confuses the variable lookup process.

We now further clarify the behavior of our interpreter: the inner bindings will *shadow* the outer ones if they happen to have the same name. (You may want to check the behavior of the `lookup` function and properly encode your context.)

Here are some test cases:

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

4.2 Multiple Bindings in Variable Declarations

Question 8. Please first write the inference rule for multiple-variable declarations (you may use any notation for lists as you like), and then modify the definition of `evaluate2` to cover the case of `DeclareMulti`.