Lab 8: Functions and Lexical Scope

This lab is due at **2359 on Thursday, 1 November**. It should be submitted to the course SI413 as Lab08, with the files named properly according to the lab instructions. See the submit page for details. You are **highly encouraged** to submit whatever you have done by the end of this lab time. Remember that **you must thoroughly test your code**; the tests that are visible before the deadline are only there to make sure you are using the right names and input/output formats.

Contents

1. Starter Code

- spl.lpp
- spl.ypp
- ast.hpp
- ast.cpp
- frame.hpp
- value.hpp
- colorout.hpp
- Makefile

You can get all these files at once by downloading the file lab08.tar.gz and running `tar xzvf lab08.tar.gz`

2. Introduction

Today we're going to implement proper lexical scoping with frames and closures in SPL. By the end of this lab, you will have working function calls with lexical scope, and will be able to run complete and powerful SPL programs.

Specifically, you will:

- Write an SPL program with a recursive function to compute Fibonacci numbers. By the end of the lab, your interpreter should be able to run

this function without any issue.

- Replace the global symbol table from last week's lab with a `Frame` object which is passed to every `eval()` and `exec()` method.
- Implement function calls which (for now) ignore arguments and return values. This requires completing the `eval` methods in the `Lambda` and `Funcall` classes of the AST.
- Implement lexical scoping by extending the `Frame` class beyond a simple symbol table, so that each Frame object contains a pointer to its parent frame, and the bind, rebind, and lookup methods search parent frames as necessary to do their work. A new frame will be created each time a Block is entered and on every function call.
- Complete the implementation of function calls with arguments and return values properly handled in their respective scopes.
- Add a new kind of "expression statement" to the SPL interpreter language, so that it's possible for example to simply call a function as a complete statement, rather than having to incorporate the function call as part of a write or assignment.

This lab will be challenging, but each individual part above only involves changing a few lines of code in your interpreter, or (in one case) making the same exact change in many places. But as we have seen, the difficulty is not in the time it takes you to write the code, but in undertanding the concepts behind what is happening so that you know what modifications are needed and where.

The starter code above is actually a solution to last week's lab, with a few changes that mostly involve cutting out parts for drawing the AST. You can start with that or with your own working solution from last week. But **if you start with your own solution, be sure to download** the new version of the `value.hpp` file, which now includes Closures, as well as updating your `main()` based on the `spl.ypp` file above. In addition, you should rename the `SymbolTable` class and corresponding file `st.hpp` from last week's lab, to call it the `Frame` class in a file `frame.hpp` instead.

3. Warm-up: Some more SPL coding!

Like last week, we are going to start by writing a small SPL program that, by the end of the lab, we should be able to correctly execute using our interpreter.

As an example of the kind of thing you need to do for this part, here is an SPL function that uses recursion to compute the value of n! - that is, n*(n-1)*(n-2)*...*1.

```
# Computes n!
new fact := lambda n {
  ifelse n <= 1
    { ret := 1; }
```

```
    { ret := n * fact@(n-1); }
};


# Read a number from the command line
new x := read;
# Print out x!
write fact @ x;
```

Notice the structure of the `lambda` in SPL: we have the keyword `lambda`, followed by the name of the SINGLE argument (all functions are unary in SPL), followed by a block of code in curly braces for the body of the function. Remember also that `ret` is a special name that must be assigned to the return value.

The way functions are called is a little unusual: we use the FUNARG operator `@`, which is defined to have *highest precedence* and is *left associative*. I assume you know what that means by now!

Exercises

1. Define the Fibonacci sequence as follows:

$$\text{fib}(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2), & n \geq 1 \end{cases}$$

   Write SPL code for a recursive function called `fib` that takes a single argument `n` and computes the n'th Fibonacci number. We define After this function definition, your program should read in a single integer `x` and print out the result of `fib@x`, that is, the xth Fibonacci number. Save your program in a file called `fib.spl`. So for example, with a complete `spl` interpreter, we should be able to do something like:

```
roche@ubuntu$ ./spl fib.spl
read> 10
55


roche@ubuntu$ ./spl fib.spl
read> 7
13
```

Run this command to submit: ~/bin/submit -c=SI413 -p=lab08 fib.spl spl.?pp ast.?pp frame.hpp value.hpp

4. Frames

Last week's lab had a single global `SymbolTable` object for the single global scope. As a first step towards implementing actual lexical scope, you will remove this global object from your interpreter, and instead pass in a `Frame*` pointer to every `eval()` and `exec()` method in the AST. In the starter code above, I've already renamed the old SymbolTable class to Frame in the file `frame.hpp`, but you need to modify the AST methods to pass and receive a frame pointer as arguments.

Specific steps:

- Remove the global symbol table declarations from the top of `ast.hpp` and `ast.cpp`
- Change the prototypes and definitions of all `eval()` and `exec()` methods to take a single `Frame*` parameter. You will have to modify where these methods are declared in each class, as well as where they are called by other AST methods. With a little bit of thought and skill with your text editor, this should be much faster than tediously going over every line of your code!
- Declare a single `Frame` in the `main()` method inside `spl.ypp`, and pass a pointer to this Frame when the AST is executed inside main. Right now, this will be the only Frame object that exists in your interpreter, but later you will create new Frames in certain key places.

Exercises

2. Change the AST so that a Frame pointer is sent to every eval and exec method, replacing the need for a single global Frame in your interpreter. This will not change the behavior of your interpreter at all compared to last week's lab, but the later tasks will be much easier by doing this first.

```
Run this command to submit:
~/bin/submit -c=SI413 -p=lab08 fib.spl spl.?pp ast.?pp frame.hpp
value.hpp
```

5. Basic functions

The next step is implementing the `eval(Frame*)` methods in your `Lambda` and `Funcall` classes so that functions actually work. **For this part, you can ignore arguments and return values**; your goal is just to be able to jump in program execution from the call site to the actual function definition in the AST.

Tips

- Evaluating a lambda is pretty simple - it just needs to return a Closure. In fact, I've created a simple `Closure` struct for you already in the `value.hpp` class, and there is a constructor to the `Value` class that creates this closure. You also might want to look at this page.

- The real fun is in `Funcall`. Because all functions in SPL have exactly one argument, there are essenitally two parts of a function call: the expression for the function itself, and the expression for the argument. Because you can ignore the argument for this part of the lab, focus on how to evaluate the function part in order to get a Value, from which you can obtain an actual Closure. That Closure should tell you what Lambda body needs to be executed, and the environment Frame in which the execution has to occur.
- Since you are ignoring the actual return values, your `Funcall::eval(Frame*)` method for now can just return `Value()`, which should print out as "UNSET" in the interpreter.

Exercises

3. Get function definitions and function calls to work, without arguments or return values. Here are a few test cases that should work at this point:

```
spl> new f := lambda x { write 20; };
spl> write f;
closure
spl> new y := f@3;
20
spl> write lambda z { new q:=13; write q*q; } @ true;
169
UNSET
```

```
Run this command to submit:
~/bin/submit -c=SI413 -p=lab08 fib.spl spl.?pp ast.?pp frame.hpp
value.hpp
```

6. Frames with Parents

Recall that a lexical frame consists of a simple symbol table (mapping of strings to values), plus a pointer to the parent frame. The `Frame` class already includes the `bindings` map for the local bindings. You need to add to this a pointer to the parent frame, then update the constructor and bind/rebind/lookup methods accordingly.

Suggested Steps

- Add a `Frame* parent;` field to the `Frame` class, and update the constructor so that it takes a frame pointer to set the parent. You will then need to update the only place in your code where a Frame is created so far, which is in the `main()` method in `spl.ypp` - the parent of this top-level Frame should be `nullptr`.
  At this point, your interpreter should still compile fine and work the same as before.

- Add the code to create a new Frame every time a `Block` is entered during execution, with the parent Frame set correctly.
  At this point, local variables should be truly local:

```
spl> new x := 10;
spl> { new x := 20; write x; }
20
spl> write x;
10
```

- Update the `lookup`, `bind`, and/or `rebind` methods of `Frame` so that they search through parent Frames as needed.
  Now nonlocal references with blocks should also work:

```
spl> new a := 100;
spl> { new x := a; write a + x; }
200
spl> write a + x;
ERROR: No binding for variable x
spl> { new y := 3; a := a + y; }
spl> write a;
103
spl> { new a := 50; a := a + 15; write a; }
65
spl> write a;
103
```

- Add the code to create a new Frame every time a `Funcall` is evaluated, with the parent of the new Frame set according to the Closure. Arguments and return values are still ignored, but this gives you quite a lot:

```
spl> new outer := 10;
spl> new f := lambda q { outer := outer * 2; };
spl> new z := f@false;
spl> write outer;
20
spl> z := f@false;
spl> write outer;
40
spl> outer := 13;
spl> z := f@false;
spl> write outer;
26
spl> { new outer := -10; z := f@z; write outer; }
```

```
-10
spl> write outer;
52
```

Exercises

4. Implement Frames with parents, including frame creation as appropriate when blocks or function calls are executed.

```
Run this command to submit:
~/bin/submit -c=SI413 -p=lab08 fib.spl spl.?pp ast.?pp frame.hpp
value.hpp
```

7. Function arguments and return values

Now it's time to actually get function calls to work. If you've done everything else correclty, all this will require is some careful footwork in the `Funcall::eval(Frame*)` method. You will need to think carefully about *when* and *in which environment* each of the following happens:

- Evaluating the argument
- Binding the argument to the parameter name
- Executing the function body
- Retrieving the return value

After this, your `fib.spl` from the beginning of the lab should work! Here's something else you can also try, which is a gcd function written recursively in SPL, using debug statements (an optional component of last week's lab) to communicate with the user:

```
new mod := lambda a { ret := lambda b {
  ret := a - (a/b)*b;
}; };


new gcd := lambda a { ret := lambda b {
  ifelse b = 0
    {ret := a;}
    {ret := gcd@b@(mod@a@b);}
}; };

"enter x and y, 0 to exit"
new x := read;
while x != 0 {
  new y := read;
  "gcd is"
```

```
  write gcd@x@y;
  "enter x and y, 0 to exit"
  x := read;
}
"goodbye"
```

Exercises

5. Get function definitions and function calls to work with proper lexical scope and correct arguments and return values.

> Run this command to submit:
> ~/bin/submit -c=SI413 -p=lab08 fib.spl spl.?pp ast.?pp frame.hpp value.hpp

8. Expression Statements

Right now, the following program should give an error:

```
new f := lambda x { write x + 5; };
f@3;
```

What's going on? Well, the problem is that a function call like `f@3` is an expression but not a statement. Now that we have function calls, we might want to have an expression as a statement all by itself, without having to `write` its result.

Your job in this part is to make this possible. This will involve adding a new subclass to the `AST` hierarchy - you might want to call it `ExpStmt`. It should be a subclass of `Stmt` that only has one part - an expression! Executing the statement will just mean evaluating the expression and throwing away the result.

Exercises

6. Allow semicolon-terminated expressions as single statements. This will require adding a new grammar rule to the `spl.ypp` Bison file as well as a new class to the `ast.hpp` file.

> Run this command to submit:
> ~/bin/submit -c=SI413 -p=lab08 fib.spl spl.?pp ast.?pp frame.hpp value.hpp

9. More Scheme in SPL (OPTIONAL)

Exercises

7. (OPTIONAL) write `cons`, `car`, and `cdr` functions for SPL. The cons should be a function that returns another function that returns a clusure (yet another function). The function returned should examine its

argument and either produce the first part of the cons pair or the second. Look closely at the gcd example above to understand how this kind of pseudo-multiple-argument function can work.
The `car` and `cdr` functions will take in a consed pair (which is a closure, remember) and evaluate the passed in function passing it a special argument that indicates which part of the pair to return.
This should be mind-blowing.
If you want to go further, think about how to implement the empty list and a predicate function `is_cons`.

Run this command to submit:
```
~/bin/submit -c=SI413 -p=lab08 fib.spl spl.?pp ast.?pp frame.hpp value.hpp cons.spl
```

8. (OPTIONAL) A significant advantage of the Scheme program above is that it is *tail-recursive*. (Remember what that means?) Because Scheme optimizes for tail recursion, the function above will only use a constant amount of space. See if you can get any tail-recursion optimization in SPL. Talk to your instructor if you have questions about how to get started. Yes, this would be very difficult to do completely. No, I don't expect anyone to complete it.