CS350 Homework 2

Due Date: September 12, 2014

This homework is about implementing an interpreter for the declarative sequential model discussed in Chapter 2 of the book.

You can assume that the source code of some Oz program is given in an easy-to-parse Abstract Syntax Tree format.

For example,

```
local X in
      X=10
end
```

is represented as

```
[localvar ident(x)
       [bind ident(x) literal(10)]]
```

Similarly

```
tree(key:10 left:nil right:nil)
```

is represented as

```
[record literal(tree)
        [[literal(key) literal(10)
         literal(left) literal(nil)
         literal(right) literal(nil)]]]
```

The full specification of the AST format is given along with each feature to be implemented.

1 Specification

A program in Oz is just a statement. Any statement in Oz is represented as a list (in Oz). In general, the source code is given as an Abstract Syntax Tree (AST), which is represented as a nested list. You have to implement an interpreter to take such an AST as input and output the sequence of execution states during the execution of the statement.

You will have to implement the semantic stack (a stack of pairs of [semantic statement, Environment]), and the single assignment store. **You are allowed to use explicit state (Cells)**. Suggested data structures are in section 3.

Code that you are provided with: Code to perform the unification algorithm (thanks to Siddharth Agarwal, a former TA for the course.) You will need this in Questions 4 and 5.

The assignment is provided in stages. A complete implementation of all the stages carries 150 points. At any stage, submit only working code. Code that does not run is ineligible for credit.

1.1 Submission Format

Submit a zip file containing your source code, and a README file containing a list of questions you have answered.

2 Questions

1. A program is a statement. The AST representation of a statement is a list.

A statement can be a nop (similar to the skip statement in Oz), or a sequence of statements. The AST of a skip statement is `[nop]`. The AST of a sequence of statements $\langle s_1 \rangle \langle s_2 \rangle$ is a list of the form `[s1 s2]`.

Implementing this part involves implementing the semantic stack - stack of pairs of [semantic statement, Environment]

(At this stage, the only statement that you have to worry about is nop. So you'll have to handle code of the form `[[nop] [nop] [nop]]`.)

(15 points)

2. Variable Creation.

```
local <X> in <s> end
```

in Oz syntax will be represented by the list `[localvar ident(x) s]`.

Implementing this feature involves implementing the "adjoining" operation on environment.

Newly Bound Occurrences: x is no longer free in this statement. (You will need a list of non-free variables if you are implementing closures in 4b).

Note that statements may be nested, so by this stage, you must take care that your code handles lexical scoping in statements such as this, correctly:

```
[localvar ident(x)
       [localvar ident(y)
              [localvar ident(x)
                     [nop]]]]
```

(15 points)

3. Variable-Variable Binding $\langle X \rangle = \langle Y \rangle$ is represented in the AST as : `[bind ident(x) ident(y)]`.

(15 points)

4. Implement the Single Assignment Store: Set of store variables, together with their bindings. A suggested data structure is the "Dictionary", as explained in Section 3. Implementing this will involve use of the unification algorithm. You can use the code provided.

The store variables are bound to values. The different kinds of values are: number, record and procedure.

a. Implement numerical values and record values.

A number is represented as follows - e.g. literal(100) (no floating point numbers)

A record is represented as a list of the following form:

```
[record literal(a)
        [[literal(feature1) ident(x1)]
         [literal(feature2) ident(x2)]
         ...
         [literal(featuren) ident(xn)]]]
```

(The second half of this question, dealing with procedures, comes later, since other features can be implemented without them.)

(15 points)

5a. Variable-Value binding to numbers and records: $\langle X \rangle = \langle v \rangle$ where $\langle v \rangle$ is a number or a record value is represented in AST as: `[bind ident(x) v]`

(15 points)

6. If-then-else

```
if <x> then <s>1 else <s>2 end
```

has AST: `[conditional ident(x) s1 s2]`

(15 points)

7. Pattern Matching.

```
case <x> of <p1> then <s>1 else <s>2 end
```

has AST:

```
[match ident(x) p1 s1 s2]
```

Newly Bound Occurrences of variables: Recall that the pattern $\langle p_1 \rangle$ is a record. All variables in the record are now bound in the case statement.

(15 points)

4 b, 5 b. Implement procedure values and assignment of variables to procedure values. A full implementation of this involves closures. You may choose to implement only those procedures with bound variables, for 15 points.

The syntax of a procedure value is

```
[proc [ident(x1) ... ident(xn)] s]
```

where x1, ..., xn are values and s is a statement. **The procedure name is not a part of the definition**.

Newly Bound Occurrences: x1,...,xn are bound in the procedure.

Free variables in the procedure: Variables which are not bound occur free. (If you implement closures, you will have to restrict the environment and store the free variable bindings as part of the procedure value.)

Procedure Definition: Variable-Value binding to procedures.

Now, extend the semantics of assignment $\langle X \rangle = \langle v \rangle$ to handle procedure values.

(30 points)

8. Procedure Application.

```
{F X1 ... Xn}
```

AST:

```
[apply ident(f) ident(x1) ... ident(xn)]
```

This involves mapping of actual parameters to formal parameters, as explained in the operational semantics.

(15 points)

2.1 Some things to note

Arithmetic expressions are not part of the syntax.

3 What you can use

To make the programming a little bit easier, you can use explicit state (CELLS).

The store can be implemented with a Dictionary. The dictionary is a standard module provided by Mozart, which is a collection of key#value tuples. You could look at Page 160 of the text to see how the Dictionary works - alternately, you could read the description here.

Mozart Version 1.4 Dictionary documentation

A code snippet that employs this module can be found here: Dict.oz. Note that the usage is different from the documentation.

In Question 4, you can use the unification code provided at Unify.oz.

4 Programming Tools

The following provides a minimal overview of the tools helpful in development. In each case, consult the Mozart Documentation for further reference.

4.1 \include directive

You may want to split the interpreter into multiple files. Oz provides a way to connect them together using the `\include` directive.

For example, in the file "Interpreter.oz", you add the statement `\include 'Unify.oz'`. Then the compiler will make all the code inside the file "Unify.oz" available in the file "Interpreter.oz".

4.2 Standalone compiler

You can compile your code outside the Interactive Environment, using the 'ozc' compiler.

```
ozc -c File.oz
```

compiles the code

```
ozc -x File.oz
```

compiles the code, and links it statically to form an executable.

4.3 Oz debugger

The oz debugger is called 'ozcar'. We recommend that you launch from the interactive environment. In the Oz interface, first select [Oz] from the menu bar, and then select [Start Debugger].

Edit the code in the interface. As usual, compile and launch the executable by clicking "Ctrl+Alt+X" at the **end** of the code. Alternatively, select the code that you want to execute using the mouse, and then select [Oz]→[Feed Region]. This should put you into debugging mode (the line executing is highlighted.)

Stepping Through Execution: To step into a line of code, use 's'. To step over, press 'n'. To continue till the next breakpoint, or to the end if there are no breakpoints, press 'c'.

Interface: We can ignore most of the features when developing sequential programs.

There are 4 windows. The "Thread Forest" window shows the threads that are executing. We have only the main thread. The "Stack of Thread" shows the call stack of the current executing code. "Local variables" shows the local variables in the topmost stack by default. "Global Variables" should be obvious.

Breakpoints: The environment provides two kinds of breakpoints.

1. To set breakpoints during execution of a program, in the editor window, go to the line where you want to set a breakpoint. Then press "Ctrl+X Space" to set a breakpoint. "Ctrl+U Ctrl+X Space" deletes a breakpoint at the current line.

2. You may instead, want to set the breakpoints before launching the debugger. To set a static breakpoint on a line, **edit the code** to have a statement {`Ozcar.breakpoint`} on the same line.

For example,

```
1. local X in
2.   X=[1 2 3]
3.   {Ozcar.breakpoint} {Browse {Append X [1]}}
4. end
```

will set a breakpoint at line 3.

Whether you have set a static or a dynamic breakpoint, start the debugger, and press 'c' twice in the debugger window to hit the next breakpoint.