Programming Assignments – 4

Programming Languages Essentials
PUCSD – January 2016-May 2016

A Simple Interpreter for a simplified λ Calculus

We use the λ calculus using the usual grammar. We do not have the ability to bind symbols to λ expressions in our programs. Instead, we must bind them before hand within the interpreter, and use them to do interpret subsequent expressions that we receive at runtime.

We use the usual list syntax. That, admittedly, mixes the underlying Scheme system, and our language that is above it. However, our language *is* list oriented! So . . . we let go :-).

The grammar is:

```
<Expression> :=    <var-ref>
                 | (lambda ({<var-ref>}*) <Expression>)
                 | (<Expression>   <Expression>)
                 | (<num-op> <Expression> <Expression>)
                 | <number>,
```

where `<number>` is any non negative integer representable in Scheme (i.e. the Scheme `read` procedure can obtain its value from an external representation at input), and `<num-op>` is one of: "+" or "$\star$". Note that "`var-ref`" is any identifier (as far as the above grammar is concerned), and "`lambda`" is a keyword. Also note that bare bones form of the application lambda term. This means that all our abstractions are in curried form!

An example of a valid expression according to the rules of the above grammar is: `(+ a b)`. An example of an invalid expression is: `(- a b)`. Expressions in the language defined by the above grammar are interpreted with respect to an environment that states the values bound to variables (i.e. identifiers). A simple representation of a binding is a 2-list (see earlier assignments), e.g. `(a 3)` is a 2-list that has the symbol `a` as its first element, and the value, `3`, bound to it as the second element. An environment can be represented by a list of such 2-lists. Before evaluating expressions in the above language, we must create an environment. Such an environment will not be able to add new bindings since our language does not have such an ability at all. It will be a static environment.

In our case, the `var-ref` *must* be one of the symbols defined by us when we create a static environment. A program with any other identifier whose lookup in the context fails is an invalid program and its interpretation is undefined.

We will name our interpreter: `our-eval-1`, the evaluator for our first language. As discussed in the class `our-eval-1` looks like: `(our-eval-1 expr-to-eval)` $\equiv$ `(reduce (lookup (parse expr-to-eval) our-env))`, where `our-env` is the static environment we must have already created.

1. Implement `parse` procedure for the above grammar. This returns an AST for valid expressions, and `#f` otherwise.
2. Implement `basic-lookup` procedure that takes a variable reference and an environment, and returns the value of bound to the variable.
3. Implement the `lookup` procedure that accepts a parse tree produced by the `parse` procedure above and an environment, and returns the parse tree with the variable references substituted by their values. You might want to use the `basic-lookup` procedure above. **Bonus**: Note that variable references may be bound to λ abstractions! This means that you may need to replace a leaf node of the parse tree by the subtree for the abstraction! You might want to do this problem in two steps: the first without adding any subtrees (e.g. λ abstractions not handled), and the second step where you handle them.
4. Implement `beta-reduce` procedure that takes a application and returns the λ expression after reducing it. Note that this only does β conversion! **Bonus**: Handle any α conversions by examining free and/or bound variables, and applying the `rename` procedure appropriately.
5. Implement `reduce` procedure that takes a valid expression (according to the grammar above), and evaluates it. In particular, it uses the `beta-reduce` procedure above where necessary.