Prof. Jingke Li (lij@pdx.edu), Lec: T 1000-1150; Labs: WR 1400-1550, R 1000-1150; Office Hour: TR 1200-1300; ZID: 87054734177

Homework 8

(Due Wednesday 3/3/21)

This homework practices object-oriented programming. It carries a total of 10 points.

In Lab 6, you worked on a compiler for a simple imperative expression language, Imp. In that compiler, an input Imp program is converted into an S-expression list first (using the `sexpr.py` module); then a compiler function, `comp()`, traverses the list to generate target stack machine code. In this homework, you are re-implementing the compiler, with two changes: (1) use classes and objects to present the input program inside the compiler, and (2) use OO-style programming to implement the compiler function. The file `impcomp2.py` is a starting version, and your task is to complete it.

Several programs from Lab 6 are copied here: `sexpr.py` (for converting input to an S-expression list), `impcomp.py` (a procedural implementation of the compiler), and `sm.py` (a target machine interpreter).

The following are the suggested steps for this homework.

1. The Imp language specification is attached at the end of this handout. If you don't remember this language, read that section to refresh yourself.
2. The file `impcomp2.py` contains a set of class definitions for representing program constructs. There are three levels of classes, e.g. `Add` is a subclass of `Binop`, which is a subclass of `Expr`. **The class structure is not be changed.** These class declarations are incomplete. Your first task is to add proper fields and constructors to the classes. Note that you should try to minimize replications, i.e. if all the subclasses have the common fields, you may want to declare them in the parent class. After completion, you can test your code by manually invoking the constructors, e.g.:

   ```
   If(Eq(Var('x'),Var('y')), Print(Add(Var('x'),Num(1))), Print(Num(0)))
   ```

3. Convert the compiler function `comp()` in `impcomp.py` to a set of `comp()` methods, one for each class. You should follow the OO programming style, and include only operations that are directly relevant to the class in its `comp()` method, and delegate children's operations to children's classes. (Recall the "dynamic dispatch" concept discussed in class this week.)
4. There is a `labelgen()` function and a `newlab` global variable in `impcomp.py`. They are for generating unique labels. Move them inside the top `Expr` class, and make `newlab` a static field.
5. Implement the function `ast(lst)` to convert an S-expression list into an `Expr` class object. This function works like a mini-compiler, translating a program construct from a list form to a class object. For example, `['+','x','y']` will be translated into `Add(Var('x'),Var('y'))`.
6. After completing everything, you should be able to run the new compiler the same way as you would with the old version:

   ```
   linux> cat sum.imp | python impcomp2.py
   ```

Submission

Submit your program `impcomp2.py` through the "Dropbox" on the D2L class website. *Important:* Keep a copy of your submission file in your folder, and do not touch it. In case there is a glitch in D2L's submission system, the time-stamp on this file will serve as a proof of your original submission time.

Appendix. Imp Language Specifications

Imp's syntax is defined by the following context-free grammar (except for `Expr` and `Op`, all symbols are terminals):

```
Expr -> num                    # integer
     |  var                    # string
     |  (Op Expr Expr)         # bin op
     |  (:= var Expr)          # assign
     |  (print Expr)           # output
     |  (seq Expr Expr)        # sequence
     |  (if Expr Expr Expr)    # if-stmt
     |  (while Expr Expr)      # while-loop
Op   -> + | - | * | / | < | =
```

Note that Imp is an imperative language, since it has imperative control constructs, `seq`, `if`, and `while`, and its expressions may have side-effects. Yet it is also an expression language because every program is a value-producing expression.

The evaluation of each expression yields a single integer result, specifically

- A number `i` yields itself.
- A variable `x` yields its current value. Every variable is implicitly initialized to 0 at the beginning of program execution.
- Evaluating `(+ e1 e2)` evaluates `e1` and then `e2`, and yields the sum of their values.

  The other arithmetic operations are similar, with one exception. For a `/` expression, if `e2`'s value is 0, the expression's value is undefined.
- Evaluating `(< e1 e2)` evaluates `e1` and then `e2`, and compares their values. If the first is less than the second, the expression yields 1; otherwise it yields 0.
- Evaluating `(= e1 e2)` evaluates `e1` and then `e2`, and compares their values. If the two values are the same, the expression yields 1; otherwise it yields 0.
- Evaluating `(:= x e)` evaluates `e`, assigns the resulting value into variable `x`, and yields that value.
- Evaluating `(print e)` evaluates `e`, prints the resulting value (followed by a `newline`) to standard output, and yields that value.
- Evaluating `(seq e1 e2)` evaluates `e1` and then `e2`, and yields the value of `e2`.
- Evaluating `(if c t f)` evaluates `c`; if the result is non-zero, then expression `t` is evaluated, otherwise expression `f` is evaluated. The `if` expression yields the value of the selected expression, either `t` or `f`.
- Evaluating `(while c b)` evaluates expression `c`; if the result is non-zero, expression `b` is evaluated and the entire `while` expression is evaluated again; otherwise the evaluation of the `while` is complete. A while expression always yields the value 0.