CSE 3341: PLAN Interpreter Project

Problem Definition

The goal of this lab is to write an interpreter for a simple functional language called PLAN. The interpreter itself should be written in Scheme. A PLAN program is a list, as defined by the following grammar; all terminals are underlined.

<Program> ::= **(prog** <Expr> **)**

<Expr> ::= <Id> |
<Const> |
(myignore <Expr> **)** |
(myadd <Expr> <Expr> **)** |
(mymul <Expr> <Expr> **)** |
(myneg <Expr> **)** |
(mylet <Id> <Expr> <Expr> **)**

<Id> ::= **a** | **b** | … | **z**

<Const> ::= **integer constant**

Here are five valid PLAN programs

- `(prog 5)`
- `(prog (myadd (myadd 7 (myignore (mymul 4 5))) (mymul 2 5)))`
- `(prog (mylet z (myadd 4 5) (mymul z 2)))`
- `(prog (mylet a 66 (myadd (mylet b (mymul 2 4) (myadd 2 b)) (mymul 2 a))))`
- `(prog (mylet x 66 (myadd (mylet x (mymul 2 4) (myadd 2 x)) (mymul 2 x))))`

Each PLAN program and expression evaluates to a particular integer value. The semantics of a program can be defined as follows:

- The entire program **(prog** <Expr> **)** evaluates to whatever <Expr> evaluates to
- **(myignore** <Expr> **)** evalutes to the integer value 0, regardless of what the subexpression <Expr> looks like
- **(myadd** <Expr> <Expr> **)** evaluates to the sum of whatever values the two sub-expressions evaluate to
- **(mymul** <Expr> <Expr> **)** evaluates to the product of whatever values the two sub-expressions evaluate to
- **(myneg** <Expr> **)** evaluates to X * (-1), where X is the integer value that the sub-expression evaluates to
- **(mylet** <Id> <Expr>$_1$ <Expr>$_2$ **)** has the following semantics. First, sub-expression <Expr>$_1$ is evaluated. The resulting integer value is “bound” to the identifier <Id>. Then the second sub-expression <Expr>$_2$ is evaluated; the result of that evaluation serves as the value of the entire **mylet** expression. The binding between the id and the integer value is active only while <Expr>$_2$ is being evaluated.

- <Id> evaluates to the value to which the identifier is bound by a surrounding **mylet** expression. If there are multiple bindings for the identifier, the last (i.e., latest, innermost) such binding is used.
- <Const> evaluates to the value of the integer constant

Based on these rules, the five programs from above are evaluated as follows:
- Program 1: evaluates to 5
- Program 2: evaluates to 17
- Program 3: evaluates to 18
- Program 4: evaluates to 142
- Program 5: evaluates to 142

Implementation

Write a Scheme function `myinterpreter` that takes as input ***a list of PLAN programs*** and produces ***a list of the corresponding values***. For example, an invocation

```
( myinterpreter
   '( (prog 5)
      (prog (mylet z (myadd 4 5) (mymul z 2)))
    )
)
```

should produce the list `(5 18)`. Your implementation must work on `scheme48` on `stdlinux`.

- You are guaranteed that the list given to the interpreter will not be empty, and will contain only valid PLAN programs. The programs will be valid both syntactically and semantically. Syntactically, you can assume that any program given is valid with respect to the grammar from above. Semantically, you can assume that any evaluation of an identifier has at least one existing binding for that identifier. For example, you can assume that the input will never contain programs of the form `(prog a)` or `(prog (mylet a 5 (myadd b 10)))`. Your implementation does not have to contain error-handling code for such cases. Do not worry about arithmetic issues such as underflow or overflow.

- Two useful library functions for your implementation are `integer?` and `symbol?`. The first one checks if its parameter is an integer constant, and the second one checks if its parameter is a symbol such as `a`, `b`, etc.

- In order to maintain the set of bindings, consider using a list where each element of the list is one specific binding. A binding is really just a pair of a symbol and an integer value.

- The only built-in Scheme functions you are allowed to use are `equal?`, `car`, `cdr`, `cons`, `cond`, `if`, `+`, `*`, `null?`, `symbol?` , and `integer?`. You should not use any other built-in function.

- Make sure your code is purely functional: do not use imperative features such as `set!`

- `(load “myfns.ss”)` inside the `scheme48` interpreter allows you to load a text file `myfns.ss` with your implementation of `myinterpreter` and any other helper functions.

Important: *A PLAN program is **not** a Scheme program. A PLAN program should **not** be directly given as input to the Scheme interpreter. Do **not** try to make the Scheme interpreter execute PLAN programs directly by defining Scheme functions `myadd`, `mymul`, etc. The PLAN program is input to your interpreter, **not** to the Scheme interpreter.*

Project Submission

On or before **11:59 pm on Wednesday, November 28** you should submit a single file that contains the definitions of all your functions, including the main function `myinterpreter`. The file itself should be called `myfns.ss`. Do not use any other name for the file or for the main function. Other functions that you define may have whatever names you choose. Use white spaces appropriately so that your function definitions are easy to read. Also, include some documentation in the same file (not a separate README file). *Comment lines* in Scheme programs start with a semi-colon.

Submit your project using the following command at the stdlinux UNIX shell prompt

```
submit c3341ac lab2 myfns.ss
```

If the time stamp on your electronic submission is **12:00 am on November 29 or later**, you will receive 10% reduction per day, for up to three days. If your submission is more than 3 days late, it will not be accepted and you will receive zero points for this project. If you resubmit your project, this will override any previous submissions and only **the latest** submission will be considered – **resubmit at your own risk**.

Grading

The project is worth 50 points. The grader will run your project with 10 test cases. The correct outputs for these test cases are worth 44 points. Additional 6 points are for code readability and documentation.

Academic Integrity

The project you submit must be **entirely** your own work. Minor consultations with others in the class are OK, but they should be at a **very** high level, without any specific details. The work on the project should be **entirely** your own: all the design, programming, testing, and debugging should be done only by you, independently and from scratch. Sharing your code or documentation with others is not acceptable. Submissions that show excessive similarities (for code or for documentation) will be taken as evidence of cheating and dealt with accordingly; this includes any similarities with projects submitted in previous instances of this course.

Academic misconduct is an extremely serious offense with **severe** consequences. Additional details on academic integrity are available from the Committee on Academic Misconduct (see http://oaa.osu.edu/coamresources.html). I strongly recommend that you check this URL. If you have any questions about university policies or what constitutes academic misconduct in this course, please contact me immediately.