

CSE 3341 Project 6

Overview

The goal of this project 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 and defined by the following grammar:

$$\begin{aligned}\langle Program \rangle &::= (\text{prog } \langle Expr \rangle) \\ \langle Expr \rangle &::= \langle Id \rangle \\ &\quad | \langle Const \rangle \\ &\quad | (\text{myif } \langle Expr \rangle \langle Expr \rangle \langle Expr \rangle) \\ &\quad | (\text{myadd } \langle Expr \rangle \langle Expr \rangle) \\ &\quad | (\text{mymul } \langle Expr \rangle \langle Expr \rangle) \\ &\quad | (\text{mysub } \langle Expr \rangle \langle Expr \rangle) \\ &\quad | (\text{mylet } \langle Id \rangle \langle Expr \rangle \langle Expr \rangle) \\ &\quad | (\text{mylet } \langle Id \rangle (\text{myfunction } \langle Id \rangle \langle Expr \rangle) \langle Expr \rangle) \\ &\quad | (\langle Id \rangle \langle Expr \rangle) \\ \langle Id \rangle &::= a \mid b \mid \dots \mid z \\ \langle Const \rangle &::= \text{integer constant}\end{aligned}$$

As examples, here are five valid PLAN programs

1. (prog 5)
2. (prog (myadd (myadd 7 (myif (mymul 4 5) 0 10)) (mymul 2 5)))
3. (prog (mylet z (myadd 4 5) (mymul z 2)))
4. (prog (mylet a 66 (myadd (mylet b (mymul 2 4) (myadd 2 b)) (mymul 2 a))))
5. (prog (mylet x 66 (myadd (mylet x (mymul 2 4) (myadd 2 x)) (mymul 2 x))))

Each PLAN program and expression evaluates to an integer value. The semantics of a program are defined as follows:

1. The entire program (prog $\langle Expr \rangle$) evaluates to whatever $\langle Expr \rangle$ evaluates to.
2. (myif $\langle Expr \rangle \langle Expr \rangle \langle Expr \rangle$) evaluates the first expression. If this is non-zero, it evaluate to the second expression, otherwise it evaluates to the third expression.
3. (myadd $\langle Expr \rangle \langle Expr \rangle$) evaluates to the sum of whatever values the two sub-expression evaluate to.
4. (mymul $\langle Expr \rangle \langle Expr \rangle$) evaluates to the product of whatever values the two sub-expression evaluate to.
5. (mysub $\langle Expr \rangle \langle Expr \rangle$) evaluates to the difference of whatever values the two sub-expression evaluate to.
6. (mylet $\langle Id \rangle \langle Expr \rangle_1 \langle Expr \rangle_2$) has the following semantics. First, $\langle Expr \rangle_1$ is evaluated. The resulting integer value is bound to the identifier $\langle Id \rangle$. Then the second sub-expression $\langle Expr \rangle_2$ is evaluated, and 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 $\langle Expr \rangle_2$ is being evaluated.

7. $\langle Id \rangle$ evaluates to the value to which the identifier has been bound by a mylet expression. PLAN will use **dynamic scoping**, i.e. if there are multiple bindings for the identifier the most recently executed active binding is used.
8. $\langle Const \rangle$ evaluates to the value of the integer constant.
9. The semantics of $(\text{mylet } \langle Id \rangle (\text{myfunction } \langle Id \rangle \langle Expr \rangle) \langle Expr \rangle)$ and $(\langle Id \rangle \langle Expr \rangle)$ are more complicated and are discussed in the next section.

Based on these rules, the five programs from above evaluate to:

1. 5
2. 17
3. 18
4. 142
5. 142

Function Semantics

When a mylet expression containing a myfunction expression like

$(\text{mylet } \langle Id \rangle_1 (\text{myfunction } \langle Id \rangle_2 \langle Expr \rangle_1) \langle Expr \rangle_2)$

is evaluated, the myfunction expression is evaluated by binding the body of the function $\langle Expr \rangle_1$ to $\langle Id \rangle_1$. This binding is only active while $\langle Expr \rangle_2$ is being evaluated.

Then, if an expression $(\langle Id \rangle_1 \langle Expr \rangle)$ is encountered while evaluating $\langle Expr \rangle_2$ and a new binding for $\langle Id \rangle_1$ has not been introduced, then the value of $\langle Expr \rangle$ is bound to $\langle Id \rangle_2$ and $\langle Expr \rangle_1$ is evaluated (once this finishes, the binding of the value of $\langle Expr \rangle$ and $\langle Id \rangle_2$ is removed). The value from evaluating $\langle Expr \rangle_1$ is the value of $(\langle Id \rangle_1 \langle Expr \rangle)$.

So for example,

1. $(\text{prog } (\text{mylet } a (\text{myfunction } b (\text{myadd } b b)) (a 5)))$ evaluates to 10
2. $(\text{prog } (\text{mylet } a (\text{myfunction } b (\text{myadd } b b)) (\text{mylet } a 1 (\text{mymul } a a))))$ evaluates to 1

Implementation

Write a Scheme function called `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)))
    )
)
```

is asking the interpreter to interpret two programs and should produce the list (5 18).

You can use an online interpreter like <https://repl.it/> for development, but for grading your implementation **must work on scheme48 on stdlinux**. A **KEY DIFFERENCE** is that the <https://repl.it/> interpreter is forgiving and will insert missing right parentheses while the scheme48 interpreter will not; if your project is working perfectly on <https://repl.it/> but then gives errors in scheme48 this is likely the problem.

Instructions and suggestions intended to help you and/or simplify your interpreter's implementation:

1. You do not need to write a scanner or a parser, the scheme interpreter automatically handles that for us.
2. Create many functions that each have a simple purpose. You do not have the ability to do sequencing, everything will have to be recursive.
3. Do not use the PLAN keywords as names your functions, i.e. instead of “(define (myadd ...)” use names like “(define (evalMyadd ...)”. This will prevent you from making a common mistake I have seen in past semesters, and should help you avoid confusion.
4. 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. **Your implementation does not have to contain error-handling code**. Do not worry about arithmetic issues such as underflow or overflow.
5. Two very useful Scheme library functions for your interpreter to use 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, ect.
6. Save handling IDs, mylet, and the list of bindings for last. In order to maintain the set of bindings, consider using a list where each element of the list is a specific binding. A binding is just a pair: the symbol and the bound value or expression.
7. Using (load “FILENAME”) or ,load FILENAME inside the scheme48 interpreter allows you to load a file named FILENAME with your implementation of myinterpreter and any other helper functions.

Instructions that limit what you can interpreter can do:

1. The **only** built-in Scheme functions you are allowed to use are
 - define, let, equal?, car, cdr, cons, list, cond, if, quote, ', +, *, null?, list?, symbol?, integer?, pair?, lambda

It is also ok to use any car/cdr variant such as cadadr. You should not use any other built-in function.

2. Make sure your code is purely function: in particular, do not use define to try to create global variables!

Project Submission

On or before **11:59 pm April 16th**, you should **submit a single file called “myfns.ss”** containing all the function definitions needed for your project, including the main function `myinterpreter`. Do not use any other name for the file or for the main function. Other functions 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 program start with a semicolon (e.g. `;this is a scheme comment`).

Submit your project to the dropbox on Carmen for Project 6.

If the time stamp on your submission is 12:00 am on April 17th or later, you will receive a 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, only the latest submission will be considered.

Grading

Your `myinterpreter` function will be tested against 20 valid test cases. The correct outputs for these test cases are worth 4 points each. An additional 20 points are for code readability and documentation. 100 points total.

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 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>). If you have any questions about university policies or what constitutes academic misconduct in this course, please contact me immediately.