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:

⟨*Program*⟩ ::= (planProg ⟨*Expr*⟩)
⟨*Expr*⟩ ::= ⟨*Id*⟩
| ⟨*Const*⟩
| (planIf ⟨*Expr*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩)
| (planAdd ⟨*Expr*⟩ ⟨*Expr*⟩)
| (planMul ⟨*Expr*⟩ ⟨*Expr*⟩)
| (planSub ⟨*Expr*⟩ ⟨*Expr*⟩)
| (planLet ⟨*Id*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩)
| (planLet ⟨*Id*⟩ (planFunction ⟨*Id*⟩ ⟨*Expr*⟩) ⟨*Expr*⟩)
| (⟨*Id*⟩ ⟨*Expr*⟩)
⟨*Id*⟩ ::= a | b | . . . | z
⟨*Const*⟩ ::= integer constant

As examples, here are five valid PLAN programs

1. (planProg 3)
2. (planProg (planAdd (planAdd 7 (planIf (planMul 4 5) 0 10)) (planMul 2 5)))
3. (planProg (planLet z (planAdd 4 3) (planMul z 2)))

4. (planProg (planLet a 66 (planAdd (planLet b (planMul 2 4) (planAdd 2 b)) (planMul 2 a))))
5. (planProg (planLet x 66 (planAdd (planLet x (planMul 2 4) (planAdd 2 x)) (planMul 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 ⟨*Expr*⟩) evaluates to whatever ⟨*Expr*⟩ evaluates to.
2. (planIf ⟨*Expr*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩) evaluates the first expression. If this is > 0 then the planIf returns the result of the second expression, otherwise it returns the result of the the third expression.
3. (planAdd ⟨*Expr*⟩ ⟨*Expr*⟩) evaluates to the sum of the values the two expressions evaluate to.
4. (planMul ⟨*Expr*⟩ ⟨*Expr*⟩) evaluates to the product of the values the two expressions evaluate to.
5. (planSub ⟨*Expr*⟩ ⟨*Expr*⟩) evaluates to the difference of the values the two expression evaluate to.
6. (planLet ⟨*Id*⟩ ⟨*Expr*⟩$_1$ ⟨*Expr*⟩$_2$) has the following semantics. First, ⟨*Expr*⟩$_1$ is evaluated. The resulting integer value is bound to the identifier ⟨*Id*⟩. Then ⟨*Expr*⟩$_2$ is evaluated, and the result of that evaluation serves as the value of the entire planLet expression. The binding between the id and the integer value is active only while ⟨*Expr*⟩$_2$ is being evaluated.

7. $\langle Id \rangle$ evaluates to the value the identifier has been bound by a planLet 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 (planLet $\langle Id \rangle$ (planFunction $\langle Id \rangle$ $\langle Expr \rangle$) $\langle Expr \rangle$) and ($\langle Id \rangle$ $\langle Expr \rangle$) are more complicated and are discussed in their own section below.

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 planLet expression contains a planFunction expression like

(planLet $\langle Id \rangle_1$ (planFunction $\langle Id \rangle_2$ $\langle Expr \rangle_1$) $\langle Expr \rangle_2$)

is evaluated, the planFunction 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. (planProg (planLet a (planFunction b (planAdd b b)) (a 5))) evaluates to 10
2. (planProg (planLet a (planFunction b (planAdd b b)) (planLet a 1 (planMul a a)))) evaluates to 1

Implementation

Write a Scheme function called myinterpreter that takes as input a single PLAN program and outputs the result of evaluating that program. For example, an invocation

```
(myinterpreter '(planProg 5))
```

should produce the output 5.

Your code must work on the scheme48 interpreter. There can be a great deal of variation in how scheme interpreters work, so please make sure you only use scheme48.

The following instructions that limit what you can do. Please ask on piazza if any of this is at all unclear:

1. The **only** built-in Scheme functions you are allowed to use are

   - define, equal?, car, cdr, cons, list, append, cond, if, and, or, not, quote, ', +, -, *, null?, list?, symbol?, integer?, pair?
   - car/cdr variants, like cadr, caar, cddr, and so on
   - let and lambda, but I do not recommend these unless you are interested in extending the project beyond what is required

   You should not use any other built-in functions.

2. Make sure your code is purely functional; here you need to give up on trying to assign to variables.

The following instructions and suggestions are intended to help you and/or simplify your interpreter's implementation:

1. Name your file "myfns.ss" and have a function named "myinterpreter" so your code interacts nicely with my scripts.

2. You do not need to write a scanner or a parser, the scheme interpreter automatically handles that for us and we will use the binary tree representation of the input as our parse tree.

3. Create many functions that each have a simple, clear purpose. You do not have the ability to do sequencing, everything will have to be recursive.

4. Do not use the PLAN keywords as names your functions, i.e. instead of "(define (planAdd ..." use names like "(define (evalPlanAdd ...". This will prevent you from making a common mistake I have seen in past semesters, and should help you avoid confusion.

5. You are guaranteed that the input given to the interpreter will not be empty, and will contain a valid PLAN program. The program 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.

6. First write code to handle Plan programs with just constants (test cases 1-5). Then add the code for handling binding ids to values (test cases 6-10). Then add the code for binding ids to functions and calling functions (test cases 11-14).

7. 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. You cannot define global variables in scheme, so you will need to pass this list as an additional parameter.

8. 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.

Testing

I have given you two files to help you with your testing.

"cases.ss" contains 14 test cases, indexed 1-14. Start your scheme interpreter and then load this file into the scheme interpreter with the command ",load cases.ss". You can then run each test case individually with the command "(test X)", where X is an integer 1-14. This will output first the expected output for the test case, and then the output from your interpreter

"TestMyfns.scm" is a script you can run from the command line (not the scheme interpreter) to run all 14 test cases.

Project Submission

On or before **11:59 pm April 21st**, you should **submit a single file called "myfns.zip"** 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 start with a semicolon (e.g. ;this is a scheme comment).**

Submit your project to the Dropbox on Carmen for Project 3.

If the time stamp on your submission is 12:00 am on April 22nd 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.