J. Anthony Timberlake, Joshua Echols
CS 401 Programming Languages
February 7, 2018

Assignment 1 : Scheme

Work Statement

The majority of the project design outside of the *apply-rules* method was developed in lab together with the assistance of the TA. Josh then implemented the first 10 cases as covered in the assignment and tested to make sure they worked as specified, including with recursive cases. Anthony then did the same for the latter 10 cases. Both tested and debugged and corresponded to make sure all tested inputs worked properly. Originally, we attempted to use a very different program design, which was much more complicated and was shortly abandoned.

Design

While the function *simplify* itself is recursive, the actual scheme file has several other methods defined which help the program to function. First, the single expression (a list) is sent to the simplify function from the command line and is checked to see if the given operator within that list is a legal operation to perform (+, *, or -). If so, the *apply-rules* function is called, wherein the 20 cases from the assignment document are spelled out and handled in 10 different conditions, some of which handle several of each of the conditions at once. If the condition passed to the *simplify* method is not a legal operation, it will be returned in itself. This way, when base cases are reached and values or symbols are evaluated, the program will know when to terminate.

Within *apply-rules*, the 10 cond's rely heavily on other functions spelled out prior, such as the functions which check for what kind of list may be present within a list (is-sum, is-prod, and is-dif). Also, the methods lhs and rhs tell us the left hand side or right hand side arguments within a given expression (list) by using cadr or caddr to extract the second and third expressions from a given list (wherein the form will always be op arg1 arg2 and arg1 and arg2 can be reached from a list within an arg# using lhs or rhs). For example, if the expression is (* 3 (+ a b)), then in this case, op = *, arg1 = 3, and arg2 = (+ a b). So, therefore, calling (lhs arg2) = a and (rhs arg2) = b. Within these methods are also helper methods which further test for what kinds of inputs are being read, meaning perhaps two constants within an expression or a constant and a term, where term is defined as either a list or a symbol and a constant is any number.

Once all of these definitions are in place, the 10 conditionals cover each of the rules and constructs new lists accordingly. In the first conditional, the base cases are handled, meaning cases 1, 3, and 5 by calling the *base* method on the three arguments of the current list being evaluated. The remaining cases are reached depending on conditions such as whether arg1 or arg2 are constants, terms, or lists and specifically if said lists are products, sums, or differences to be further evaluated. In all of those cases except for rules 2, 4, and 6, all of the rules are recursively called with *simplify* surrounding the list construction of each independent list case. This will make sure that once each value of lists within lists have been evaluated, they will also be put into the correct form as per each case.

Difficulties

Originally, the program design was to be constructed of a nested loop structure, meaning it did not follow based on each individual case, which quickly led to complicated spaghetti code that was very hard to test and follow along with. Additionally, the very large number of parentheses and simplistic nature of if-else conditionals resulted in confusing scenarios when building onto code in many instances as well.

Pros of this Language

Scheme does a good job of displaying the differing layers of recursion that are being dealt with. The overall design of the language is very similar to a giant mathematical formula, with parentheses for precedence operations. It makes sense in that it similarly formats which operations should be pushed onto the stack for a recursive call first. In that sense, this language is the best I've seen for showcasing recursion and its properties in a concise manner with no frills or nonsense added, such as needless key words in places one would expect there to be something more. The car/cdr functions were also more helpful than just trying to use list-ref or standard array based indexing. Once one could play with it a little more, the language itself feels lightweight and easy to learn new things, expressing things a multitude of ways, especially with the use of the REPL.

Cons of this Language

Because of the simplistic nature of nested conditionals and if statements, it can be overwhelming at first to determine what is happening and what is the determined else statement (as it is whatever comes in the following parenthetical pair). Additionally, it was quite an adjustment to enforce the placement of operations in front of the two operands in a given list. The biggest con is probably that we will never likely use this language in our professional careers and because of that, finding resources to assist in learning the language or resolving issues within the language are not as widespread as more well-known computer languages. It also means that becoming proficient in scheme has no real marketability or practical application other than the fact that it's somewhat neat and different from what one may have grown to expect from programming languages seen thusfar.