Homework 9

Robert Steele

Problem 2)

I chose to do problem two because it was the easier of the two. And this question really was very easy.

Syntax functions:

I started by defining some helper functions in syntax.scm like cond-clauses which retrieves the clauses from a cond expression, no-more-clauses? which checks to see if there are clauses that can still be evaluated, first-clause which get the first clause from a list of clauses and rest-clauses which retrieves the rest of the clauses from a list of clauses. I also wrote cond-actions and cond-predicate to give me access to the actions and predicates of a cond clause.

Order of execution:

My order of execution for evaluating conds was as follows. Retrieve all the clauses of the cond expression. If there are no clauses remaining, we exit and return some throwaway value (in this case false). Otherwise check if the next clause is an else clause. If it is, then we check to make sure it is the last clause and if it is then we evaluate the actions and finish. If it is not the last clause then we throw an error. If it is not an else clause then we retrieve the actions and predicate of the clause and evaluate the predicate. If the predicate evaluates to true, then we evaluate the actions and finish execution. If the predicate is false, then we loop to evaluate the next clause. (check to make sure there is another clause etc…) To accomplish this I wrote some helper functions to retrieve all the information I needed (clauses, first clause, second clause, predicate, actions) and two other expressions to check for specific cases (no more clauses?, else clause?). Once I wrote these, I added a few execution blocks with their own labels to jump to. The blocks are as follows. Ev-cond begins evaluation of the cond statement. Ev-cond-clause evaluates the next available conditional clause (checks to make sure there is one and that it is not an else). Ev-cond-dispatch decides what to do once the predicate has been evaluated. Ev-cond-actions evaluates the actions associated with a cond clause. Ev-cond-else handles else clauses. Ev-cond-false handles the case where there are no more clauses to evaluate. And finally, ev-exit-cond restores continue and resumes execution at that registers value.

Final thoughts:

This was a fun problem. I enjoy having to account for pushing and popping to stacks and like the way everything lines up when it is done properly. While I did not really have any problems that took me a long time to fix it was a learning curve to translate the obvious solution to a register-based format. Most of my time was spent reorganizing what was saved and where to find the cleanest way to write everything.

Problems 3 and 4)

Iterative Factorial w/ tail recursion

n	total-pushes	max-depth
1	64	10
2	99	10
3	134	10
4	169	10
5	204	10

Recursive Factorial w/ tail recursion

n	total-pushes	max-depth
1	16	8
2	48	13
3	80	18
4	112	23
5	144	28

The functions to calculate the maximum depth and total number of pushes for both the iterative and recursive factorial for all n >= 1 are listed in the table below.

	Total Number of Pushes	Maximum Depth
Iterative Factorial	29 + 35n	10
Recursive Factorial	32n - 16	3 + 5n

These values show that with tail recursion enabled iterative functions have a constant stack space requirement whereas recursive functions require a linear amount relative to their input. It also shows that the recursive function requires fewer stack pushes per recursion than iterative functions require per iteration. This means that at least in this case while recursive functions require more space, they complete their tasks in a shorter amount of time.

Problem 5)

Iterative Factorial w/out tail recursion

n	total-pushes	max-depth
1	70	17
2	107	20
3	144	23
4	181	26
5	218	29

Recursive Factorial w/out tail recursion

n	total-pushes	max-depth
1	18	11
2	52	19
3	86	27
4	120	35
5	154	43

The functions to calculate the maximum depth and total number of pushes for both the iterative and recursive factorial for all n >= 1 after removing tail recursion are listed in the table below.

	Total Number of Pushes	Maximum Depth
Iterative Factorial	33 + 37n	14 + 3n
Recursive Factorial	34n - 16	3 + 8n

These values show that removing tail recursion causes iterative functions to require address space linear in relation to their input n rather than constant space. Recursive functions will also require more stack space for each recursive call. In this case recursive functions still require more space than its iterative counterpart and still has a faster processing time as well.