COMP-301 FALL-2021 PROJECT-3 REPORT

Project Group Members:
1-) Lütfü Mustafa Kemal Ato (KU ID Number: 69579)
2-) Barış Kaplan (KU ID Number: 69054)

All the parts are working without an error.

Part-1:

In part 1, our main approach is to increment counter at every call-exp, so that we can increment the counter in every procedure call. We use scheme built-in functions begin and set! to increment the counter by 1. Then, we display the counter with the given format in the pdf file with the display function.

Part-2:

We first changed the lang.scm. We added the grammatical specifications of the proc-nested-exp, call-nested-exp, and call-nested-exp accordingly as specified in the pdf.

Then, we did the translation in the translator.scm as follows: we translated proc-exp to the proc-nested-exp, we gave var, 'count, 'anonym, and (translation-of body env) to the proc-nested-exp. 'anonym and the 'count values are for the procedure name and the call count. We translated call-exp to call-nested-exp, our first approach was to increment the count in this call-nested-exp, we planned to do this by doing cases on the rator. If it was a const-exp, then this means that it has been initialized, and the count can be incremented here. If it is not a const-exp, then the count could be initialized. But then we did the incrementation in interp.scm (we asked if we could do it this way, and we learned that it is okay to do the incrementation in the interp.scm). So in the final version of our call-nested-exp does the translation of rator with env, and translation of rand with env, and gives the count as (const-exp1). "else case" does the same as the "const-exp case". We translated the letrec-exp to the letrec-nested-exp, we gave p-name (procedure name), b-var, 'count, (translation-of p-body env), and (translation-of letrec-body env) to the letrec-nested-exp.
In the data-structures.scm, we have added a nested-procedure to the proc define-datatype. It takes 2 additional parameters compared to the normal procedure. The first parameter is the name parameter. Name parameter is a symbol type. It holds the name of the procedure. The second parameter is the cont parameter, which is also a symbol type. It holds the number of recursive calls to the procedure. We also added a new environment type named extend-env-rec-nested to the define-datatype environment. It takes 1 additional parameter compared to extend-env-rec, which is count. The type of count is symbol. It holds the number of recursive calls to the procedure.

In the interp.scm, we added the behaviour specifications in the value-of for the proc-nested-exp, call-nested-exp, letrec-nested-exp. For the proc-nested-exp, it takes var, count name, and body, and it returns a proc-val type nested-procedure with the parameters var, name, body, count, and env. For the call-nested-exp, it takes rator, rand, count. With the help of let, it stores a procedure, which is the value-of rator in env, turned to a procedure from expval with the help of the function expval->proc, and this is stored as proc. There is also arg, which stores the arguments, we get the arguments by calling value-of on rand with env. Then, we return (apply-procedure proc arg), which applies the procedure with the given arg (arguments).
Moreover, we added the case for nested-procedure in the apply-procedure. If the apply-procedure is called with nested-procedure (in the case where proc1 is nested-procedure), we first get bvar, name, body, count, and env.
Then, we call recursive-displayer with the name (name from the nested-procedure), and for the num parameter of the recursive-displayer, we do the following: We get the count with apply-env, turn it to a integer from expval with expval->num function, add 1 to it, turn the added value to a num-val, extend the env with variable 'count and the value of the addition in the env. Then we do apply-env to count in env to get the latest count variable.

We also call the value-of on body with the environment which is first extended with the new count variable (count is incremented by one as we did in the recursive-displayer part.) ; then, extended with extend-env-rec-nested given name (procedure name), bvar, body, count. Finally, the environment is extended with bvar and arg.
In the environment.scm, we first added extend-env with 'count and (num-val 0) to the beginning of init-env lambda. Then for the apply-env, for the extend-env case, we changed it as follows: if val is equal to 'proc, then we return proc-val nested-procedure. For getting the parameters of nested-procedure, we do cases, we take bvar, id, body, count, env. Next, we return proc-val nested-procedure with bvar, var (changed from id to var, var is from the extend-env's parameter), body, count, env. Otherwise, if it is not a procedure we return the val.
We also added the case for extend-env-rec-nested, which takes the parameters id, bvar, body, count, saved-env. Then the process is similar to extend-env-rec; instead of returning a procedure, we return nested-procedure with arguments nested procedure, bvar, id, body, count, env.

Additional test cases: We have written 3 additional test cases; a test for nested-procs, a test for let & letrec, and a test for custom sum-until procedure.

Part-3:

The translation of var-exp is as follows: we translate it again to var-exp, but we give (apply-senv-number senv var) as an argument. (this gives us the desired format: name + number of occurences)

What apply-senv-number does is as follows: it extends the senv with var (notice var is cons'ed with '(), so it is given as a pair). Then, we call apply-senv with arguments; the extended environment, and var, and store the result of the apply-senv in val1. After this, by using symbol->string, we convert var to string (this var is from the argument of the apply-senv-number). Similarly, by using number->string, we convert val1 to string. Then we append them with string-append function. Finally, by using string->symbol, we converted the appended string to a symbol and returned it.

let-exp with the arguments var, exp1, body is translated to let-exp with the arguments (apply-senv-number senv var) (this gives us the desired format: name + number of occurences), (translation-of exp1 senv), and (translation-of body (extend-senv var senv))

At the beginning of the translation of proc-exp, we call (extend-senv var senv) to update the senv so that we get the correct occurences. After that, we translate proc-exp with the arguments var, body to proc-exp with the arguments (apply-senv-number senv var), and (translation-of body (extend-senv var senv)).

Workload Breakdown: We did all the parts together.