Project 3 Report
COMP 301 - Spring 2022
May 20, 2022
Group 13: Özkan Yamaner - 69740 & Murat Güç - 68967

Part 1

Since we need to count the number of occurrences of variable initializations, and as it was given as a hint, we created a global variable named *"count"* to do that job. We initialized it just outside of the translation-of procedure.

In let-exp part of the translation-of procedure, we check whether that variable of let statement is already in the environment. We do it by using the includes? function on variable and the static environment. If var is not in the environment, this means a variable initialization and we increment the "count" variable (count of new variable initializations.). If var is already in the environment, it means that we are re-initializing the same variable name. In that case, counter does not change and we display a message that says the variable is re-initialized, with the name of the variable that got re-initialized.

Part 2

lang.scm: We added the grammar details of the newly defined expressions in "the-grammar" part of the lang.scm. We did it according to the abstract and concrete grammar specifications given in the project handout.

data-structures.scm: In the data-structures.scm, we added some new data types which are "nested-procedure" to handle nested procedures, under proc data types and "extend-env-rec-nested" under environment data types. In addition to the normal procedure, nested-procedure has two new parameters, name of procedure and count variable. Furthermore, extend-env-rec-nested data type has additional 1 parameter for count which is a symbol.

environments.scm: In environments.scm file, we extended the initial environment by adding the variable "n" for the counter, whose initial value is 0. In the extend-env part under the apply-env, val is checked to see whether it is a nested-procedure or not. We used the "cases" function to detect it. If it is a nested-procedure, then proc-val with a nested-procedure will be returned. Otherwise, extend-env behaves as it normally does. In the extend-env-rec-nested part, we reorganized and augmented extend-env-rec function to work with nested-procedure.

interpreter.scm: In our interpreter, as usual, we implemented the new nested expression types under the procedure "value-of". They are "proc-nested-exp", "call-nested-exp" and "letrec-nested-exp". As before, we took the standard versions of these expression types as a basis and modified them according to the new requirements. "proc-nested-exp" calls the "nested-procedure" with its extra parameters instead of the "procedure" utility of the standard version. "Call-nested-exp" gets the count as an additional parameter. As a difference from the

standard version, it pairs-up its operand (arg) with the count and passes it to “apply-procedure” accordingly. “letrec-nested-exp” is pretty similar to the standard version and it calls “extend-env-rec-nested” instead of “extend-env-rec” with the count as a new parameter.

We also extend the “apply-procedure” to accommodate for new additions. It has another procedure type (“nested-procedure”) in its “cases” statement. It evaluated the procedure body while accounting for the new count parameter. It also uses the given helper procedure called “recursive-displayer” to print the recursive call stack of recursive procedures with the procedure name (if the procedure has a name)

translator.scm: In translator.scm, we translate the original let-exp, proc-exp, and call-exp to nested procedure versions. We implement these under the “translation-of” procedure. While normal proc-exp is taking parameters which are variable and body, proc-nested-exp has a name which is initialized as “anonym” and the count which is “n”. These are temporary initial symbols. Translation of call-exp handles a specific situation. If the operator input in the call-exp is var-exp, that means that variable is already in the environment. Therefore, its count value located in the environment will be incremented by 1. If the operator is not var-exp, it initializes its count as 1 by sending the parameter of const-exp to the call-exp. In the translation of letrec-exp to letrec-nested-exp, count is included by sending “n” to the nested version as a parameter.

Part 3

In this part, it is asked that variable names should be together with its occurence. First, we need to define apply-senv-number to count variables in the environment. Procedure is implemented recursively, it scans each variable in the environment and questions whether it is equal to the search variable. Base case is the null situation of list, it returns 0 in the context of summation. Then, if the first element of the environment list is equal to the search-var, it increments by 1 and accumulates thanks to the recursion, which means it calls itself by sending the parameters that are the rest of the environment and the search-var. If the first element is not equal to the search-var then it recursively searches the remaining part of the environment.

After implementing the apply-senv-number procedure which returns the occurrence of a variable, we started to handle translations of var-exp, let-exp, and proc-exp. Implementation of var-exp should return the variable itself i.e “x” and its occurrence i.e “3” by concatanating them like “x3”. So, to do that, first get the variable and convert it from symbol to string, then get the number from apply-senv-number procedure then convert number to string. After converting everything to string then we used the string-append procedure and converted the last result to symbol. Let-exp and proc-exp include the similar implementation like var-exp while handling variable in them but let-exp and proc-exp add new variables to the environment so they should increment occurrence of the variable by 1 after calling apply-senv-number.

All tests were passed and gave correct results.

Workload Breakdown

Özkan mainly implemented Part 1 while Murat mostly focused on Part 3. We helped each other when one of us struggled. Because, Part 2 was the longest and the most complicated, we worked on it together all the time.