COMP-301 PROJECT-4

Project Partners:

Barış Kaplan (KU ID NUMBER: 69054)
Lütfü Mustafa Kemal Ato (KU ID NUMBER: 69579)

Workload Breakdown: We did all the parts together.
Note: All parts of the project work properly.

Part-A:

Changes in data-structures.scm:

We have added a new case for expval, it is arr-val which is a list of references.
We have defined expval->arr, it does cases on the input, if it is an arr-val, we extract the array (named as array-2), and return it. If it is not an arr-val, use expval-extractor-error with 'array array to print the error message. We have defined create-array, it takes two arguments; length and values. It has a recursive helper function defined inside it with letrec. If length is zero, return '() so that it can be cons'ed with the last element. If length is not zero, cons the newref of value with the value that comes from the function being recursively called (decrement length by 1 when calling it recursively).

Changes in lang.scm:

We have added the expressions to the lang.scm with the format specified in the pdf.

Changes in interp.scm:

Behavior specification of newarray-exp takes two arguments: length and value. It does value-of on the length, then turns it to int from num-val. It calls value-of on value. Then it returns arr-val of create-array with these arguments.

Behavior specification of update-array-exp: It takes 3 arguments: array, index, value. It takes the value-of array. Turns it to arr from expval and stores it as v1. Then it does value-of to the index and turns it to an int from expval. It also does value-of to the value. Then it calls setref! with list-ref (v1 & v2), and v3.

Behavior specification of read-array-exp: It takes two arguments, array, and index. It gets the arr value of the array, and it gets the length of the array. It creates a recursive function with letrec. The recursive function takes an array, index, and end. If the index is equal to the end, it terminates. Otherwise, it prints the elements of the array using deref on the index.

Behavior specification of print-array-exp: It takes one argument, array. It gets the arr value from it and stores it in arr2 with let. It stores the length of the arr2 with let again. Then it creates a recursive function with letrec. The recursive function is as follows: it takes an array, index, and

end. If the index is equal to the end, it terminates. Otherwise, it derefs to the index, turns it to a num, and displays it. Then it calls the recursive function again with index incremented by one.

Part-B:

Our stack works as follows: It uses arrays. Stack is actually an array, with the first index (0th index) being the size of it. All the values initialized as -1024. So basically we call create-array with arguments 1024 and -1024.

Changes in interp.scm:

Behavior specification of newstack-exp: We first call create-array with the arguments 1024 and -1024. Then we set the first index to 0, since it stores the size. Then we return the arr-val of it.

Behavior specification of stack-push-exp: It takes two arguments: stack and value. It gets the stack as an arr value with expval->arr, and the new value that will be pushed which is value-of on the value (given as argument). Then it sets the new value to the top with setref!. Finally, it increments the 0th index of the stack, which holds the size, with setref!.

Behavior specification of stack-pop-exp: It takes one argument, stack. It gets the stack as an arr value with expval->arr, and size. After that, it stores the popped element as popped-element. The popped element is deref on the size of the stack. Then it checks if the size is 0 or not. If the size is 0, it returns -1. Otherwise, it setref!'s the popped index to -1024 (as implemented). Then it decrements the size of the array by calling setref! with the arguments list-ref on stack & 0, and size decremented by 1. Finally, it returns the popped-element that we have stored before.

Behavior specification of stack-size-exp: It takes one argument, stack. It gets the stack as an arr value with expval->arr. Then it uses deref on the 0th element of the stack to get the size of the stack and returns it.

Behavior specification of stack-top-exp: It takes one argument; stack. It gets the stack as an arr value with expval->arr. It gets the size of the stack by calling deref on the 0th element of the stack. Then it calls deref again on the stack with this value (size).

Behavior specification of empty-stack?-exp: It takes one argument; stack. It gets the stack as an arr value with expval->arr. It gets the size of the stack by calling deref on the 0th element of the stack. If this value is equal to zero, it returns #t, else #f.

Behavior specification of print-stack-exp: It takes one argument; stack. It gets the stack as an arr value with expval->arr. It creates a recursive function in it with letrec. The recursive function takes two arguments: stack and index. If the index is equal to the size of the stack, it displays the last element. Otherwise, it displays the current element by calling deref on the list-ref on the

stack with the index. Then it calls the helper function again with the parameters stack and index incremented by 1.

Changes in lang.scm:

We have added the expressions to the lang.scm with the format specified in the pdf.

Part-C:

Changes in interp.scm:

Behavior specification of array-comprehension-exp: It takes three arguments: body, var, and array. It turns the array into arr val and stores it with let. Then it stores the length in len. It creates a procedure with the arguments var body and env. Var and body come from the arguments and env come from the value-of. Then it turns this procedure to a proc-val, then it turns it into a procedure with expval->proc and stores it with let. After that, we define a recursive helper function with letrec. The recursive functions take the following arguments: array to store arr-val, index to do iteration, and end to keep the end of the list. If the index reaches the end, we return the arr val. Otherwise, in a begin statement, we first call setref! with list reference on arr with the index, and we call-apply-procedure with the arguments: the procedure that we have stored in let, and the deref on list reference on arr with the index. Then we call the helper function recursively with the arguments arr, the index incremented by one, and end.

Changes in lang.scm:

We have added the expressions to the lang.scm with the format specified in the pdf.