Project 3
COMP301 Fall 2024

Deadline: December 27, 2024 - 23:59 (GMT+3 : Istanbul Time)

In this project, you will work in groups. You may either use the same groups as the previous project, or modify the assignments into new groups using the link below:

Link to Google Sheets for Choosing Group Members

There is a single code boilerplate provided to you: `Project3MyProc`. Submit Racket files to LearnHub as a zip. Name your submission files as:

p3_member1IDno_member1username_member2IDno_member2username.zip
Example: *p3_0011221_jpinkman22_0011222_wwhite21.zip*

The deadline for this project is December 27, 2024 - 23:59 (GMT+3 : Istanbul Time). **Read your task requirements carefully. If you have any questions, don't hesitate to use the discussion board. Good luck!**

Table 1. Grade Breakdown for Project 3

Question	Grade Possible
Part A	50
Part B	50
Total	100

Problem Definition: As you have seen in the previous project, it is possible to implement a custom programming language using Scheme. A language can be created by specifying its syntax and behavior according to a given interface. In this project, you will extend the PROC language which you have seen in class to add more functionalities. The new language will be called MyProc. Its syntax is defined as follows:

Program ::= *Expression*

a-program (exp1)

Expression ::= *Number*

const-exp (num)

Expression ::= -(*Expression* , *Expression*)

diff-exp (exp1, exp2)

Expression ::= *zero?* (*Expression*)

zero?-exp (exp1)

Expression ::= if *Expression* then *Expression*
{elif *Expression* then *Expression*}*
else *Expression*

if-exp (exp1 exp2 conds exps exp3)

Expression ::= *Identifier*

var-exp (var)

Expression ::= let *Identifier* = *Expression* in *Expression*

let-exp (var exp1 body)

Expression ::= proc (*Identifier*) *Expression*

proc-exp (var body)

Expression ::= (*Expression Expression*)

call-exp (rator rand)

Expression ::= empty-stack()

stack-exp()

Expression ::= stack-push(*Expression*, *Expression*)

stack-push-exp (stack, exp)

Expression ::= stack-pop(*Expression*)

stack-pop-exp (exp)

Expression ::= stack-peek(*Expression*)

stack-peek-exp (exp)

Expression ::= stack-push-multi(*Expression*, {*Expression*}*)

stack-push-multi-exp (stack, exps)

Expression ::= stack-pop-multi(*Expression*, *Expression*)

stack-pop-multi-exp (stack, num)

Expression ::= stack-merge(*Expression*, *Expression*)

stack-merge-exp (stack1, stack2)

FIGURE 1. Syntax for the MyProc language

Part A (50 pts). Implement the MyProc language given in Figure 1 by adding the missing expressions.

(1) (12.5 pts) Add *stack-exp* to the language. *stack-exp* should take no arguments and should return an empty stack. The stack can be represented as a list in the Scheme language.

(2) (12.5 pts) Add *stack-push-exp* to the language. For MyProc, we assume that the stack accepts only numbers. *stack-push-exp* should take a stack and a number that will be added to the top of the stack.

(3) (12.5 pts) Add *stack-pop-exp* to the language. *stack-pop-exp* should take a stack as input and return a stack also but with the element at the top of the stack removed. If the input stack is already empty, it should return an empty stack, but print an accompanying warning message.

(4) (12.5 pts) Add *stack-peek-exp* to the language. *stack-peek-exp* takes a stack as input and returns the element in the top of the stack. If the stack is empty, the expression must return the number 2813 which will denote a failed operation and also print a warning message.

In the given code, we have provided the implementations for basic Proc that we have seen in the class, and your task is to implement the rest.

Part B (50 pts).

(1) (17.5 pts) Add *stack-push-multi-exp* to the language. *stack-push-multi-exp* takes a stack as input, as well as an arbitrary number of expressions (which evaluate to a number each), and adds them all to the stack in the order in which they were given. For example, if the command `stack-push-multi(s, 1, 2, 3)` is evaluated, 1 will be added to the stack first, then 2, then 3 and so on.

(2) (17.5 pts) Add *stack-pop-multi-exp* to the language. *stack-pop-multi-exp* takes a stack and a number n as output. It returns a stack after removing n elements from the top of the stack. If n is greater than the size of the stack, all the elements are removed and a warning message must be printed.

(3) (15 pts) Add *stack-merge-exp* to the language. *stack-merge-exp* takes two stacks as input and pops the elements of the second stack and pushes them one by one to the first stack. This means that the order in which the elements of the second stack are pushed to the first one, is the same order in which they are popped from the second one. It returns a single stack representing the merge operation of both stacks.

Note 1: We provided several test cases for you to try your implementation. Uncomment corresponding test cases and run `tests.rkt` to test your implementation.

Note 2: For the sake of passing the tests, it is important to implement the stack such that each new element is added and popped from the leftmost side of the stack.

Note 3: Usage of `arbno` in the Grammar

In the provided *lang.rkt* file, the `arbno` is used to specify that certain elements can be repeated zero or more times.

For example:

- In the definition of the `identifier`, `arbno` is used to allow for an arbitrary number of letters, digits, underscores (_), hyphens (), or question marks (?). This allows for flexible identifiers like `x`, `var_name`, or `expr1`.

- In the `number` rule, `arbno` is used to define a sequence of digits, making it possible to represent any number of arbitrary length.

Note 4: Usage of `begin` expression in Scheme

In cases where you need to display error messages and return values later, you should ensure the error message is shown first, followed by the return value. In Scheme, you can use the begin expression to evaluate a series of sub-expressions in sequence. Please refer to this page for examples on how to use begin; it could be helpful for your task.

Link to Racket Reference documentation about Sequencing