Sabancı University
Faculty of Engineering and Natural Sciences

CS305 Programming Languages

Homework 5

Due: 29 December 2021, 11:55 PM

1 Introduction

In this homework you will implement a Scheme interpreter, similar to the ones we saw in the lectures. However, the subset of Scheme that is handled by the interpreter will be larger.

2 The Scheme subset `s7`

The syntax of the Scheme subset that will be covered by the interpreter that you will implement for this homework is given in this section.

We have already implemented a sequence of interpreters for Scheme during the lectures. You can find these interpreters in the lecture notes and on SUCourse.

If you like, you can take the most advanced interpreter we have on SUCourse, and modify it to implement the additional features required by this homework. On the other hand, if you like you can also start implementing a brand new interpreter yourself.

Grammar for the subset s7

```
              <s7> -> <expr>
                    | <define>

            <expr> -> NUMBER
                    | IDENT
                    | <if>
                    | <let>
                    | <lambda>
                    | <application>

          <define> -> ( define IDENT <expr> )

              <if> -> ( if <expr> <expr> <expr> )

             <let> -> ( let ( <var_binding_list> ) <expr> )

          <lambda> -> ( lambda ( <formal_list> ) <expr> )

     <application> -> ( <operator> <operand_list> )

        <operator> -> <built_in_operator>
                    |  <lambda>
                    |  IDENT

<built_in_operator> -> + | * | - | /

    <operand_list> -> <expr> <operand_list>
                    |

<var_binding_list> -> ( IDENT <expr> ) <var_binding_list>
                    |

     <formal_list> -> IDENT <formal_list>
                    |
```

Note that anything that is accepted as a number by "number?" predicate is a number for our interpreter as well. This is how we have been implementing the numbers in the interpreters we developed in the class. Compared to the subsets we handled in the class, this grammar allows much more liberal expressions to be used: we will be able to apply a lambda expression directly within an interaction as

```
((lambda (n) (+ n 2)) 5)
```

and we will also be able bind a lambda expression to a variable as

```
(define inc2 (lambda (n) (+ n 2)))
```

and apply it later as

```
(inc2 5)
```

Note that, if there is a variable in procedure, it checks the value when the procedure is applied. You can see the example for binding variables to some values in Section 3.

We are now also able to use "if" expressions.

```
(if ((lambda (n) (+ n 2)) 2) (+ 1 2) (- 3 5))
```

Since we now have the "if" expression, we actually need boolean values as well. However, to keep things simple for the homework, we will represent boolean values by numeric values. We will adopt the following approach:

- Number 0 is considered as the boolean false.
- Any number value other than 0 is considered as the boolean true.

You can assume that any expression that will be used in the "if" (where normally a boolean value is expected) will be a number. You don't need to check if it is really a number and you don't need to produce an error if it is not a number.

We defined above an "if" expression to have 3 expressions. This is the normal syntax. The first expression is the "test expression". The second expression is the "then expression" (the value of "then expression" is used when the value of the "test expression" is true, i.e. a non-zero numeric value). The third expression is the "else expression". The value of "else expression" is used when the value of the "test expression" is false (i.e. 0).

You should pay attention to the semantics of “let”. In the following sequence of expressions

```
(define x 5)
(let ((x 3)(y x)) (+ x y))
```

the “let” expression should produce the value 8.

If we are given the following let expression where the same variable `x` has more than one binding:

```
(let ((x 3)(x 1)) (+ x 3))
```

the “let” expression should produce an error. In other words, a “let” expression cannot have multiple bindings for the same variable.

A note on the number of items in `<operand_list>` in an `<application>`.

- If the `<operator>` is a lambda expression, then the number of items in the `<operand_list>` and the number of formal parameters in the lambda expression must match. If they are different, then an error should be produced.
- When the `<operator>` is the addition, multiplication, subtraction or division operators, you can assume that the `<operand_list>` will always contain two or more items. Your code will not be tested with cases that contain less than 2 items in its `<operand_list>`. Therefore, you do not need to handle such conditions (e.g. `(+), (/ 2), (-)` etc.).
- The division and subtraction operators are left associative. You can assume that division by 0 will not be performed. You don’t need to check for this condition or produce an error.

Your interpreter should check the syntax of the expression to be interpreted. If the syntax is not correct, then an error message should be displayed. If the syntax is correct, then the value of the expression will be displayed.

Below, we provide some sample interactions in “Scheme Interaction” part, which we hope would explain the semantics of the constructs a little bit more. You can also see the format of the error and value messages to be displayed in this part.

3 The procedure `cs305`

You should declare a procedure named `cs305` which will start the interpretation when called. It should not take any arguments.

In every iteration of your REPL, you should print out the prompt given below in "Scheme Interaction" sample, then accept an input from the user, then evaluate the value of the input expression, and finally print the value evaluated by using a value prompt. The following is a sample on how the interaction with your interpreter must look like.

Scheme Interaction

```
1 ]=> (cs305)
cs305> 3
cs305: 3

cs305> (define x 5)
cs305: x

cs305> x
cs305: 5

cs305> ((lambda (n) (+ n 2)) 5)
cs305: 7

cs305> (define inc2 (lambda (n) (+ n 2)))
cs305: inc2

cs305> (inc2 5)
cs305: 7

cs305> (define incx (lambda (n) (+ n x)))
cs305: incx

cs305> (incx 1)
cs305: 6
```

Scheme Interaction cont'd

```
cs305> (define x 1)
cs305: x

cs305> (incx 1)
cs305: 2

cs305> (define x 5)
cs305: x

cs305> x
cs305: 5

cs305> (define y 7)
cs305: y

cs305> y
cs305: 7

cs305> (+ x y)
cs305: 12

cs305> (+ x y (- x y 1) (* 2 x y ) (/ y 7))
cs305: 80

cs305> (if x (+ x 1) (* x 2))
cs305: 6

cs305> (if (- 5 x) (+ x 1) (* x 2))
cs305: 10

cs305> (define z (let ((x 1) (y x)) (+ x y)))
cs305: z

cs305> z
cs305: 6
```

Scheme Interaction cont'd

```
cs305> (let ((x 1)) (+ x y z))
cs305: 14

cs305> (let () (+ x y z))
cs305: 18

cs305> (define 1 y)
cs305: ERROR

cs305> (def x 1)
cs305: ERROR

cs305> t
cs305: ERROR

cs305> (if 1 2)
cs305: ERROR

cs305> (if 1)
cs305: ERROR

cs305> (if)
cs305: ERROR

cs305> (let (x 3) (y 4) (+ x y))
cs305: ERROR

cs305> (let ((x)) (+ x y))
cs305: ERROR

cs305> (let (()) (+ x y))
cs305: ERROR
```

In the example, `incx` increments the argument `n` by `x`. `x` is evaluated when the procedure is called. `x` is not bound when the procedure is defined, but when the procedure is applied/called.

As you may recall, when an error is detected, the MIT Scheme Interpreter and the interpreters we developed in the class, produce an error message and go into an error prompt.

However, in this homework, as you may have noticed in the interaction given above, when there is an error, you interpreter should display an error message and **it should go back to "the read prompt" again**, not into another error prompt. Hence, we will be able to interact with the interpreter even after there is an error in the expression.

In order to do this, you can simply use the built-in "display" procedure for displaying the error messages (not the "error" procedure we used in the interpreters we developed in the class).

4 How to Submit

Submit your Scheme file named as `username-hw5.scm` where `username` is your SUNet username. We will test your submissions in the following manner. A set of test cases will be created to asses the correctness of your scheme interpreter.

Each test case will be automatically appended to your file. Then the following command will be executed to generate an output. Then your output will be compared against the desired output.

```
scheme < username-hw5.scm
```

So, make sure that the above command is enough to produce the desired output.

5 Notes

- **Important**: Name your files as you are told and **don't zip them.** [-10 points otherwise]

- **Important: Make sure your procedure name is exactly cs305**
- **Important: Since this homework is evaluated automatically make sure your output is exactly as it is supposed to be.**
- **Important**: SUCourse's clock may be off a couple of minutes. Take this into account to decide when to submit.
- No homework will be accepted if it is not submitted using SUCourse+.
- You may get help from our TA or from your friends. However, **you must implement the homework by yourself**.
- Start working on the homework immediately.
- Note that, you may be able to find Scheme interpreters for Windows. Although it is discouraged, you may use them. However, we want to remind you that, your homework will be evaluated on `flow.sabanciuniv.edu`. Hence we recommend that you, at least, test your implementation on this before submitting.
- LATE SUBMISSION POLICY:
  Late submission is allowed subject to the following conditions:
  - Your homework grade will be decided by multiplying what you get from the test cases by a "submission time factor (STF)".
  - If you submit on time (i.e. before the deadline), your STF is 1. So, you don't lose anything.
  - If you submit late, you will lose 0.01 of your STF for every 5 mins of delay.
  - We will not accept any homework later than 500 mins after the deadline.
  - SUCourse+'s timestamp will be used for STF computation.
  - If you submit multiple times, the last submission time will be used.