Sabancı University
Faculty of Engineering and Natural Sciences

CS305 Programming Languages

Extra Homework 2

Due: 22-05-2020 @ 23:55

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 below.

We have already implemented a sequence of interpreters for Scheme during the lectures. You can find these interpreters on SUCourse (see Resources page). If you like, you can simply take the most advanced interpret we have on SUCourse and modify it to implement the additional fetaures required by this homework. On the other hand, if you like you can also start implementing a brand–new interpreter of yourself.

The last subset of Scheme we handled in the class was named as `s6`. Let us call the subset we will handle in this homework as `s7`. The subset `s7` has `if`, `cond`, `let`, and `let*` as the new constructs to be added on top of `s6`. Therefore, it would be sufficient for you if you take the interpreter for `s6` from SUCorse, and simply add the implementation of `if`, `cond`, `let`, and `let*`.

As a matter of fact, we have explained in the class, after you implement `let`, implementing `let*` can easily be handled by converting `let*` expressions into equivalent `let` expressions. See the lecture notes for the explanation of how one can convert a `let*` expression into an equivalent `let` expression. Similarly, the implementation of `cond` expression can be performed by converting a given `cond` expression into an equivalent `if` expression (i.e. into a sequence of nested `if` expressions), and interpreting this `if` expression.

Grammar for the subset `s7`

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

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

            <expr> -> NUMBER
                    | IDENT
                    | <if>
                    | <cond>
                    | <let>
                    | <letstar>
                    | ( <procedure> <actuals> )

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

            <cond> -> ( cond <conditional_list> <else_condition> )

<conditional_list> -> <conditional>
                    | <conditional> <conditional_list>

     <conditional> -> ( <expr> <expr> )

  <else_condition> -> (else <expr>)

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

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

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

       <procedure> -> +
                    | -
                    | *
                    | /

         <actuals> -> <expr> <actuals>
                    |
```

Note that anything accepted as a number by `number?` predicate is a number for our interpeter as well. This is how we have been implementing the numbers in the interpreters we developed in the class.

Since we now have `if` and `cond` expressions, 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.

We would like to emphasize the following points about the syntax and semantics of the new constructs that you will include into the subset in this homework.

- 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).
- Note that, normally an `if` expression can be used with 2 expressions only. Such an `if` expression would not have an "else expression". However, to keep things simple in this homework, we restrict ourselves to `if` expressions with all 3 expressions existing. This is in fact how we presented the grammar above.
- A `cond` expression will have a non–empty list of alternative expressions (as represented by `<conditional_list>` above). In addition, there will always be an "default alternative" at the end (as represented by tt ¡else_condition¿ above). Note again that normally, a `cond` expression may have an empty list of "alternative expressions" and it may not have any "default alterntive". However, again to keep things simple in this homework, we restrict the syntax of `cond` expressions as described above.
- The semantics of a `cond` expression is exactly the same as we considered in the class. We start by considering the first alternative (i.e. `<conditional>`), and traverse the alternatives one by one. Each alternative has two expressions. The first expression is the "test expression", the second expression is the "value expression". When we consider an alternative, if the "test expression" evaluates to true (i.e to a non–zero numeric value), the value of the "value expression" is returned as the value of the entire `cons` expression. If the "test expression" evaluates to false (i.e to 0), then we proceed and consider the next alternative. If this process reaches to the default alternative at the end, the value of the expression that we have in the default alternative is returned as the value of the entire `cond` expression.
- `let` and `let*` expressions can have empty list of bindings. The semantics of `let` and `let*` expression are the same as we explained in the class.

Below, we provide some sample interactions in "Scheme Interaction" part, which we hope would explain the semantics of the contructs a little bit more.

3 The procedure `cs305`

You should declare a procedure named `cs305` which will start the show 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> (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> (cond
         (0             (+ x 1))
         ((* y 0 )      (/ y 0))
         ((- y (+ x 5)) (* x y))
         (else               -1)
       )
cs305: 35
```

Scheme Interaction (cont'd)

```
cs305> (cond
         (0             (+ x 1))
         ((* y 0 )      (/ y 0))
         ((- y (+ x 2)) (* x y))
         (else               -1)
       )
cs305: -1

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

cs305> z
cs305: 6


cs305> (define z (let* ((x z) (y x))
                         (if (- (/ y 2) 3) (+ x z) (- z (/ x 3)))))
cs305: z

cs305> z
cs305: 4

cs305> (let ((x 1)) (+ x y z))
cs305: 12

cs305> (let () (+ x y z))
cs305: 16

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
```

Scheme Interaction (cont'd)

```
cs305> (cond)
cs305: ERROR

cs305> (cond 1)
cs305: ERROR

cs305> (cond 1 2)
cs305: ERROR

cs305> (cond 1 2 3)
cs305: ERROR

cs305> (cond (1 2) (1 3) (2 4))
cs305: ERROR

cs305> (cond (1 2) (else 3) (else 4))
cs305: ERROR

cs305> (cond (else 3) (3 4))
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
```

The `ERROR` message can be anything you want to write. However make sure that it appears on the same line as the value prompt `cs305:`

Use the built–in `display` procedure for displaying the error messages.

4 How to Submit

Submit only one file (**without zipping it**) on SUCourse. The name of your file must be:

username-extra2.scm

where username is your SUCourse username.

5 Notes

- **Important**: Name your file as you are told and **don't zip it.**
  [-10 points otherwise]
- 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 write the file you will submit yourself**.
- Start working on the homework immediately.
- We will have STF (Submission Time Factor) for late submissions (0.01 drop in STF for each 5 mins of delay after the deadline).