Jelle in Racket

Introduction

The quickest route to implementing a new language is usually to write an interpreter. In this project, you will implement an interpreter for jelle using the Racket implementation of Scheme. This project serves to give you further experience both with language implementation and with programming in a functional language. To make this process as simple as possible, we will implement a "schemified" version of Jelle that preserves the essence of our language while adapting its syntax so that programs can be embedded directly as Scheme data.

RJelle Description

Basic Form

RJelle uses the structures of Jelle embedded directly in Scheme lists. Extraneous punctuation is removed. Here is our example quadratic equation solving program written in the new form.

```
(
  (display "Enter the value of a: ")
  (input a)
  (display "Enter the value of b: ")
  (input b)
  (display "Enter the value of c: ")
  (input c)

  (discrim := (- (^ b 2) (* (* 4 a) c)))
  (discroot := (^ discrim 0.5))

  (display "root1 is" (/ (+ (- b) discroot) (* 2 a)) nl)
  (display "root1 is" (/ (- (- b) discroot) (* 2 a)) nl)
  )
```

Notice that a program is now simply a list of statements where each statement is itself a list. Also, expressions are now done with fully parenthesized prefix notation so that our program looks a lot like an AST. This will allow us to interpret programs without a separate parsing step.

Language Extensions

As discussed in class, to "complete" our language, we will need some control structures. We are adding a simple while loop and an if statement with an optional else. Here is the form for the if:

```
(if <condition> (<if-statements>) <else-statements>)
```

Here's our program with a simple (one-way) decision to prevent it from bombing on equations that have no real roots:

```
(
  (display "Enter the values of  a b c: ")
  (input a b c)
  (discrim := (- (^ b 2) (* (* 4 a) c)))
  ( if (>= discim 0)
     ( (discroot := (^ discrim 0.5))
       (display "root1 is " (/ (+ (- b) discroot) (* 2 a)) nl)
       (display "root1 is " (/ (- (- b) discroot) (* 2 a)) nl)
       )
     )
  )
```

A two-way decision simply adds in statements after the if block:

```
(
  (display "Enter the values of  a b c: ")
  (input a b c)
  (discrim := (- (^ b 2) (* (* 4 a) c)))
  ( if (>= discrim 0) (
       (discroot := (^ discrim 0.5))
       (display "root1 is " (/ (+ (- b) discroot) (* 2 a)) nl)
       (display "root2 is " (/ (- (- b) discroot) (* 2 a)) nl)
       )
       (display "Sorry!" nl)
       (display "This equation has no real roots!" nl)
       )
  )
```

What does a multi-way decision look like?

```
(
  (display "Enter the values of  a b c: ")
  (input a b c)
  (discrim := (- (^ b 2) (* (* 4 a) c)))
  (if (> discrim 0) (
      (discroot := (^ discrim 0.5))
      (display "root1 is " (/ (+ (- b) discroot) (* 2 a)) nl)
      (display "root2 is " (/ (- (- b) discroot) (* 2 a)) nl)
      )
  (if (= discrim 0)(
      (root := (/ (- b) (* 2 a)))
      (display "Double root at " root nl)
      )
      (display "No real roots!" nl)
  ))
)
```

Note: This is a bit awkward since we need enough closing parentheses to match all of the `ifs`.
And here's our simple while loop:

```
(while <condition> <statements>)
```

Let's add an interactive loop to our code.

```
(
  (continue := 1)
  (while (= continue 1)
      (display "Enter the values of  a b c: ")
      (input a b c)
      (discrim := (- (^ b 2) (* (* 4 a) c)))
      (if (> discrim 0) (
          (discroot := (^ discrim 0.5))
          (display "root1 is " (/ (+ (- b) discroot) (* 2 a)) nl)
          (display "root2 is " (/ (- (- b) discroot) (* 2 a)) nl)
          )
      (if (=  discrim 0)(
          (root := (/ (- b) (* 2 a)))
          (display "Double root at " root nl)
          )
          (display "No real roots!" nl)
      ))
      (display "Solve another? (1 = yes, 0 =no) ")
      (input continue)
      )
   (display "Thanks for using the qaudratic solver!" nl)
   (display "Please consider supporting the dedicated programmers" nl)
   (display "who created this app!" nl)
)
```

These and other example programs can be found in the file `testprogs.scm`.

Assignment

You can find the basic outline of a Jelle interpreter in the file `jele_interp.scm`. This program is complete enough to execute `prog0` and the first line of `prog1` found in `testprogs.scm`. You can try this out in DrRacket.

```
> (run prog0)
Hello, World!
'()
> (run prog1)
1 = 1
. . Unimplemented expression type (+ 1 2)
>
```

Notice that the `run` function executes the program and also returns the final contents of memory. Hence `(run prog0)` returns an empty list because it had no input or assignment statements.

The initial interpreter only works for programs consisting of display statements and expressions that are numeric literals. You need to complete this interpreter so that it can run any legal Jelle program. The test programs provide increasingly complex examples that you can use for motivating incremental work on the interpreter.

Don't forget that you can also test each function interactively. For example, you can test out the `eval-expr` by hand by doing something like:

```
>(eval-expr '(+ a b) test-mem)
3
```

(See the Memory ADT cade at the top to see the contents of `test-mem`)

IMPORTANT NOTE: If you want to define your own simple test programs, feel free to do so. You will need to "export" the id for your program at the top of the file (see the `provide` at the top of `testprogs`). Then you will need to import the file at the top of the interpreter (see `require` at the top of `jelle_interp.scm`.