CmpE 260 - Principles of Programming Languages
Spring 2022 - Project 2

Deadline: 30 May 2022 23:59

Assistants: Alper Ahmetoğlu, Hazar Çakır
`ahmetoglu.alper@gmail.com`, `hazar.cakir.cmpe@gmail.com`

1   Introduction

In this project, you are going to implement an interpreter that will translate a statement in *Yerel-Hesap* language written in infix notation [1] to a statement in prefix notation (also known as Polish notation) [2]. This will help us converting arithmetic statements that we use in our daily lives to statements that can be executable by the Racket interpreter. A very simple example is shown below:

```
(Infix notation)
3 + 4 * 7 = 31
(3 + 4) * 7 = 49
(Prefix notation)
(+ 3 (* 4 7)) = 31
(* (+ 3 4) 7) = 49
```

Notice that if you can convert an expression in infix notation into a prefix one, you can directly evaluate it in the Racket interpreter:

```
> (eval '(+ 3 (* 4 7)))
31
> (eval '(* (+ 3 4) 7))
49
```

2   Language definition

We will formally define our language as a context free grammar to eliminate any ambiguity regarding the possible sentences in the language (i.e., valid arithmetic expressions). Production rules are as

follows:

$$\text{Expression} \rightarrow \texttt{Expression + Expression} \tag{1}$$
$$\text{Expression} \rightarrow \texttt{Expression * Expression} \tag{2}$$
$$\text{Expression} \rightarrow \texttt{(Expression)} \tag{3}$$
$$\text{Expression} \rightarrow \texttt{(BindingList) @ (Expression)} \tag{4}$$
$$\text{Expression} \rightarrow \texttt{number} \tag{5}$$
$$\text{Expression} \rightarrow \texttt{variable} \tag{6}$$
$$\text{BindingList} \rightarrow \texttt{BindingList BindingList} \tag{7}$$
$$\text{BindingList} \rightarrow \texttt{Assignment --} \tag{8}$$
$$\text{Assignment} \rightarrow \texttt{'variable := 'variable} \tag{9}$$
$$\text{Assignment} \rightarrow \texttt{'variable := number} \tag{10}$$

where `number` can be any valid number in Racket (e.g., `3`, `2`, `3.14`, `3e-4`) and `variable` is a single letter symbol (e.g., `a`, `b`, `c`, ... `z`).

There are some special rules that set expressions in this language a little different from what we commonly use. Namely, Rule 4 introduces local scoping for variables: any `Assignment` in `BindingList` is only valid in the expression right after `@`.

2.1 Example expressions from the language

- `3 + 4` (the result is 7)

- `5`

- `3 * 4 + 7` (19)

- `3 * (4 + 7)` (33)

- `a + 7` (a valid expression, but your program should raise error if you try to evaluate)

- `('a := 4 --) @ (a + 7)` (11)

- `a + ('a := 4 --) @ (a + 7)` (the first `a` is not in the scope, a valid expression, but the program will raise error if you try to evaluate)

- `('a := 7 --) @ (a * ('a := 4 --) @ (a + 7))` (now this is both a valid expression, and can be evaluated, the result is 77)

- `1 + 7 + 6 + 1 + 34 * (('x := 3 -- 'y := 6 --) @ (5 * (x + y + ('z := 'x --) @ (z)))) + 5` (2060)

- `((((((6))))))` (the result is 6)

- `('a := 1 -- 'b := 2 -- 'c := 3 -- 'd := 4) @ (('a := 95 --) @ (('d := 47 --) @ (('d := 60 --) @ (a) + ('b := 11 --) @ (69) + 38 + c + 85 + ('b := 64 --) @ (('c := 89 --) @ ((('d := 29 --) @ (('c := 65 --) @ (73)))) + 26) * 22 * ('a := 65 --) @ (c + b) * ('a := 16 --) @ (2) * b * 7 * 98 + 73 * 96 + c + 88 + ('c := 46 --`

```
'd := 58 -- 'a := 13 --) @ (('d := 82 --) @ (('d := 45 --) @ ((d + d) * 52 + ('b
:= 30 --) @ (81))) * ('a := 71 --) @ ((('d := 8 --) @ (d * 25 * c * ((('d := 12
--) @ (b))) + 32))) * 42) + 18 * (c) + ('c := 34 -- 'a := 98 --) @ (80) + 87 +
61 + 72)) + 36 * (((b) * 61)) + (25) + 17 * (a)) (the result is 3715593921)
```

You are provided with a script, generate_sentence.py, that randomly samples valid sentences from this language. However, some sentences might not be evaluatable (if a symbol is not binded to any number).

3 Functions

You are going to implement six functions that will help you to parse and evaluate statements in this language.

3.1 (:= *var value*) 10 points

Given a variable symbol var and a number value value, this function will return var and value in a list. This function will be useful for parsing assignments.

Examples:

```
> (:= 'x 1337)
'(x 1337)
> (:= 'y 260)
'(y 260)
```

3.2 (-- *assignment1 assignment2 ... assignmentN*) 10 points

This function will translate multiple variable assignments in this language into binding part of the let function, which would in turn help for parsing local assignments and expressions. Namely, this function constructs a new list with the first element being the keyword let and the rest being the concatenated assignments.

Examples:

```
> (-- '(x 3) '(y 14))
'(let ((x 3) (y 14)))
> (-- (:= 'x 15) (:= 'y 92))
'(let ((x 15) (y 92)))
> (-- (:= 'x 65) (:= 'y 35) (:= 'z 89))
'(let ((x 65) (y 35) (z 89)))
```

3.3 (@ *ParsedBinding Expr*) 10 Points

Given a parsed binding list (i.e., the output of --) and an expression, this function returns a valid let statement. You should be able to evaluate the result of this function (assuming the expression is valid).

Examples:

```
> (@ '(let ((x 5))) '(x))
'(let ((x 5)) x)
> (@ '(let ((x 5) (y 4) (z 3))) '((* x y z)))
'(let ((x 5) (y 4) (z 3)) (* x y z))
> (eval (@ '(let ((x 5) (y 4) (z 3))) '((* x y z))))
60
> (@ (-- (:= 'x 5) (:= 'y 4) (:= 'z 3)) '((* x y z)))
'(let ((x 5) (y 4) (z 3)) (* x y z))
```

3.4 (split_at_delim *delim args*) 20 points

Given a list of arguments `args`, this function splits `args` by the delimiter `delim` and returns a list of partitions where each partition holds arguments in between two `delim`s.

Examples:

```
> (split_at_delim 'a '(1 2 3 a 4 5 a 7 123))
'((1 2 3) (4 5) (7 123))
> (split_at_delim '+ '(1 2 + 3 4))
'((1 2) (3 4))
> (split_at_delim '+ '(1 + (2 + 3) + 4))
'((1) ((2 + 3)) (4))
> (split_at_delim '+ '(1 * (2 + 3) + 4))
'((1 * (2 + 3)) (4))
> (split_at_delim '* '(1 * (2 + 3) + 4))
'((1) ((2 + 3) + 4))
> (split_at_delim '* '(666 123))
'((666 123))
```

3.5 (parse_expr *expr*) 30 points

This is the main function that transforms expressions in *YerelHesap* to expressions in Racket. Once you implement previously defined functions, this function returns a valid Racket expression which you can evaluate to get a result.

Examples:

```
> (parse_expr '(1 + 3))
'(+ 1 3)
> (parse_expr '(1 + x))
'(+ 1 x)
> (parse_expr '(3 + 4 * 7))
'(+ 3 (* 4 7))
> (parse_expr '(3 * 4 + 7))
'(+ (* 3 4) 7)
```

```
> (parse_expr '(3 * (4 + 7)))
'(* 3 (+ 4 7))
> (parse_expr '(((((3 + 4))))))
'(+ 3 4)
> (parse_expr '(3))
3
> (parse_expr '(('x := 5 --) @ (x + 1.618)))
'(let ((x 5)) (+ x 1.618))
> (parse_expr '(('x := 5 --) @ (('x := 6 --) @ (x * x))))
'(let ((x 5)) (let ((x 6)) (* x x)))
> (parse_expr '(('x := 5 --) @ (x * ('x := 6 --) @ (x))))
'(let ((x 5)) (* x (let ((x 6)) x)))
> (parse_expr '(('x := 5 --) @ (y * ('x := 6 --) @ (x))))
'(let ((x 5)) (* y (let ((x 6)) x)))
> (parse_expr '(1 + 7 + 6 + 1 + 34 * (('x := 3 -- 'y := 6 --) @ (5 * (x + y + ('z :=
'x --) @ (z)))) + 5))
'(+ 1 7 6 1 (* 34 (let ((x 3) (y 6)) (* 5 (+ x y (let ((z x)) z)))))) 5)
> (parse_expr '(('a := 1 -- 'b := 2 -- 'c := 3 -- 'd := 4) @ (('a := 95 --) @ (('d :=
47 --) @ (('d := 60 --) @ (a) + ('b := 11 --) @ (69) + 38 + c + 85 + ('b := 64 --)
@ (('c := 89 --) @ ((('d := 29 --) @ (('c := 65 --) @ (73)))) + 26) * 22 * ('a := 65
--) @ (c + b) * ('a := 16 --) @ (2) * b * 7 * 98 + 73 * 96 + c + 88 + ('c := 46 --
'd := 58 -- 'a := 13 --) @ (('d := 82 --) @ (('d := 45 --) @ ((d + d) * 52 + ('b :=
30 --) @ (81))) * ('a := 71 --) @ ((('d := 8 --) @ (d * 25 * c * ((('d := 12 --) @
(b))) + 32))) * 42) + 18 * (c) + ('c := 34 -- 'a := 98 --) @ (80) + 87 + 61 + 72))
+ 36 * (((b) * 61)) + (25) + 17 * (a))))


'(let ((a 1) (b 2) (c 3) (d 4))
   (+
    (let ((a 95))
      (let ((d 47))
        (+
         (let ((d 60)) a)
         (let ((b 11)) 69)
         38
         c
         85
         (*
          (let ((b 64)) (+ (let ((c 89)) (let ((d 29)) (let ((c 65)) 73))) 26))
          22
          (let ((a 65)) (+ c b))
          (let ((a 16)) 2)
          b
          7
          98)
         (* 73 96)
         c
```

```
      88
      (let ((c 46) (d 58) (a 13))
        (*
         (let ((d 82)) (let ((d 45)) (+ (* (+ d d) 52) (let ((b 30)) 81))))
         (let ((a 71)) (let ((d 8)) (+ (* d 25 c (let ((d 12)) b)) 32)))
         42))
      (* 18 c)
      (let ((c 34) (a 98)) 80)
      87
      61
      72)))
  (* 36 (* b 61))
  25
  (* 17 a)))
```

(spaces, tabs, linebreaks might not be the same in your terminal)

3.6 (eval_expr *expr*) 20 points

This function first parses an expression `expr`, then evalutes it and returns the result. Once you implement `parse_expr`, the implementation of this function is trivial.

Examples:

```
> (eval_expr '(1 + 3))
4
> (eval_expr '(1 + x))
```
(You should get an error saying that x is undefined. You don't have to implement the error output. Just make sure that the above expression raises an error.)
```
> (eval_expr '(3 + 4 * 7))
31
> (eval_expr '(3 * 4 + 7))
19
> (eval_expr '(3 * (4 + 7)))
33
> (eval_expr '(((((3 + 4))))))
7
> (eval_expr '(3))
3
> (eval_expr '(('x := 5 --) @ (x + 1.618)))
6.618
> (eval_expr '(('x := 5 --) @ (('x := 6 --) @ (x * x))))
36
> (eval_expr '(('x := 5 --) @ (x * ('x := 6 --) @ (x))))
30
> (eval_expr '(('x := 5 --) @ (y * ('x := 6 --) @ (x))))
Error
```

```
> (eval_expr '(1 + 7 + 6 + 1 + 34 * (('x := 3 -- 'y := 6 --) @ (5 * (x + y + ('z :=
'x --) @ (z)))) + 5))
2060
> (eval_expr '((('a := 1 -- 'b := 2 -- 'c := 3 -- 'd := 4) @ (('a := 95 --) @ (('d :=
47 --) @ (('d := 60 --) @ (a) + ('b := 11 --) @ (69) + 38 + c + 85 + ('b := 64 --)
@ (('c := 89 --) @ ((('d := 29 --) @ (('c := 65 --) @ (73)))) + 26) * 22 * ('a := 65
--) @ (c + b) * ('a := 16 --) @ (2) * b * 7 * 98 + 73 * 96 + c + 88 + ('c := 46 --
'd := 58 -- 'a := 13 --) @ (('d := 82 --) @ (('d := 45 --) @ ((d + d) * 52 + ('b :=
30 --) @ (81))) * ('a := 71 --) @ ((('d := 8 --) @ (d * 25 * c * ((('d := 12 --) @
(b))) + 32))) * 42) + 18 * (c) + ('c := 34 -- 'a := 98 --) @ (80) + 87 + 61 + 72))
+ 36 * (((b) * 61)) + (25) + 17 * (a))))
3715593921
```

4 Submission

You will submit two files: `main.rkt`, `feedback.txt`. Your code should be in one file named
`main.rkt`. First four lines of your `main.rkt` file must have exactly the lines below since it will be
used for compiling and testing your code automatically:

```
; name surname
; student id
; compiling: yes
; complete: yes
```

The third line denotes whether your code compiles correctly, and the fourth line denotes whether
you completed all of the project, which must be **no** if you are doing a partial submission. This
whole part must be lower case and include only the English alphabet. Example:

```
; alper ahmetoglu
; 2012400147
; compiling: yes
; complete: yes
```

We are interested in your feedback about the project. In the `feedback` file, please write your
feedback. This is optional, you may leave it empty and omit its submission.

5 Prohibited Constructs

The following language constructs are ***explicitly prohibited***. You *will not get any points* if you
use them:

- Any function or language element that ends with an !.

- Any of these constructs: `begin`, `begin0`, `when`, `unless`, `for`, `for*`, `do`, `set!-values`.

- Any language construct that starts with `for/` or `for*/`.

6 Tips and Tricks

- You can use higher-order functions `apply`, `map`, `foldl`, `foldr`. You are also encouraged to use anonymous functions with the help of `lambda`.

- You can use Racket reference, either from DrRacket's menu: Help, Racket Documentation, or from the following link `https://docs.racket-lang.org/reference/index.html`.

- A useful link for the difference between `let`, `let*`, `letrec`, and `define`: `https://stackoverflow.com/questions/53637079/when-to-use-define-and-when-to-use-let-in-racket`.

References

[1] Infix notation. `https://en.wikipedia.org/wiki/Infix_notation`.

[2] Polish notation. `https://en.wikipedia.org/wiki/Polish_notation`.