How to Design a programming language (Simplified Python)

Programming Language Design Course

Lecturer: Mohammad Izadi

Authors:
Mahdi Saber
Mehdi Lotfian
Parsa Enayati

Contents

Section 1: implementing the lexer and parser

All programs written in our language, need to use proper syntax and should be able to make a program tree. In fact, executing a program is also executing this tree and returning the result. But how can we make a tree like this? The answer is, the lexer and parser. So in the first step we should design a lexer and parser for our language to make such tree. In this section we first talk about lexers and than we go through the whole process of designing a parser.

1.1: what is lexer?

The first thing we should do is to take the written program (in string format) and split that to different tokens. Lexer will do the job for us. So what lexer do exactly? Before we answer this question, let's talk about tokens. Tokens are defined in racket[1] and can be used to implement the lexer.

We have tokens, and empty-tokens. Empty tokens are predefined strings like "+", "return" and "def". "+" will always be PLUS token. Normal tokens – in the other hand - need an input like a variable name "myAge" or a number like "1.93".

For our implementation, we'll use these tokens (you can see the implemented syntax in page TODO):

```
(define-tokens x (NUM ID))
(define-empty-tokens y (EOF SEMICOLON PASS BREAK CONTINUE
ASSIGNMENT RETURN GLOBAL DEF OP CP COLON COMMA TRUE FALSE IF ELSE
FOR IN OR AND NOT EQUALS LT LET GT GET PLUS MINUS TIMES DIVIDES
POWER OB CB NONE))
```

Lexer will take an string and matches the string with different regex and when it finds a match, it'll make its token and returns it.

for example, imagine we have the following string:

```
for x     in      my_list:
      if x >= 5: continue
      powers = powers + x**2
```

we expect that the lexer will split this code to the following tokens:

FOR, (ID "x"), IN, (ID "my_list"), COLON, IF, (ID "x"), GET, (NUM 5), COLON, CONTINUE, (ID "powers"), ASSIGNMENT, (ID "powers"), PLUS, (ID "x"), POWER, (NUM 2), EOF

[1] The language we used to implement the whole project.

as you can see, we will completely ignore the whitespace characters.

So now we can implement the lexer as below:

```
(define full-lexer (lexer
   (whitespace (full-lexer input-port))
   ((:or (:: (:? #\-) (:+ (char-range #\0 #\9))) (:: (:? #\-) (:: (:+ (char-range #\0 #\9)) #\. (:+ (char-range #\0 #\9))))) (token-NUM (string->number lexeme)))
   ((:+ (:or (char-range #\0 #\9) (char-range #\a #\z) (char-range #\A #\Z) #\_)) (token-ID lexeme))
   ((eof) (token-EOF))
   (";" (token-SEMICOLON))
   ("pass" (token-PASS))
   ("break" (token-BREAK))
   ("continue" (token-CONTINUE))
   ("=" (token-ASSIGNMENT))
   ("return" (token-RETURN))
   ("global" (token-GLOBAL))
   ("def" (token-DEF))
   ("(" (token-OP))
   (")" (token-CP))
   (":" (token-COLON))
   ("," (token-COMMA))
   ("if" (token-IF))
   ("else" (token-ELSE))
   ("for" (token-FOR))
   ("in" (token-IN))
   ("or" (token-OR))
   ("and" (token-AND))
   ("not" (token-NOT))
   ("==" (token-EQUALS))
   ("<" (token-LT))
   ("<=" (token-LET))
   (">" (token-GT))
   (">=" (token-GET))
   ("+" (token-PLUS))
   ("-" (token-MINUS))
   ("*" (token-TIMES))
   ("/" (token-DIVIDES))
   ("**" (token-POWER))
   ("[" (token-OB))
   ("]" (token-CB))
   ("True" (token-TRUE))
   ("False" (token-FALSE))
   ("None" (token-NONE))))
```

notice that we used these operators to match the input string:

- The exact pattern (::)

- One of the following patterns (:or)
- One or none match(es) of the pattern (:?)
- One or more matches of the pattern (:+)