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: The Grammar of language

In this section we want to design a grammar for our language. We are going to design a programming language like python – a simplified version of it – through this paper. So the first step is to design a proper grammar for our language. The implemented grammar for the first parts of the paper is shown below (note that we change this grammar gradually):

1. program → Statements EOF
2. Statements → Statement ';' | Statements Statement ';'
3. Statement → Compound_stmt | Simple_stmt
4. Simple_stmt → Assignment | Global_stmt | Return_stmt | 'pass' | 'break' | 'continue'
5. Compound_stmt → Function_def | If_stmt | For_stmt
6. Assignment → ID '=' Expression
7. Return_stmt → 'return' | 'return' Expression
8. Global_stmt → 'global' ID
9. Function_def → 'def' ID '(' Params ')' ':' Statements | 'def' ID '(' ')' ':' Statements
10. Params → Param_with_default | Params ',' Param_with_default
11. Param_with_default → ID '=' Expression
12. If_stmt → 'if' Expression ':' Statements Else_stmt
13. Else_stmt → 'else' ':' Statements
14. For_stmt → 'for' ID 'in' Expression ':' Statements
15. Expression → Disjunction
16. Disjunction → Conjunction | Disjunction 'or' Conjunction
17. Conjunction → Inversion | Conjunction 'and' Inversion
18. Inversion → 'not' Inversion | Comparison
19. Comparison → Eq_sum | Lt_sum | Let_sum | Gt_sum | Get_sum | Sum
20. Eq_sum → Sum '==' Sum
21. Lt_sum → Sum '<' Sum
22. Let_sum → Sum '<=" Sum
23. Gt_sum → Sum '>' Sum
24. Get_sum → Sum '>=' Sum
25. Sum → Sum '+' Term | Sum '-' Term | Term
26. Term → Term '*' Factor | Term '/' Factor | Factor
27. Factor → '+' Power | '-' Power | Power
28. Power → Atom '**' Factor | Primary
29. Primary → Atom | Primary '[' Expression ']' | Primary '(' ')' | Primary '(' Arguments ')'
30. Arguments → Expression | Arguments ',' Expression
31. Atom → ID | NUMBER | List | 'True' | 'False' | 'None'
32. List → '[' Expressions ']' | '[' ']'
33. Expressions → Expressions ',' Expression | Expression

Section 2: 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 after designing the grammar, we should design a lexer and parser for our language to make such tree. In this section we first talk about lexers and then we go through the whole process of designing a parser.

2.1: 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 section 1):

```
(define-tokens a (NUM ID))
(define-empty-tokens b (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))
```

Figure 2.1 - all required tokens for our lexer

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
```

Figure 2.2 - a simple for loop in Simplified python

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

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

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

Figure 2.3 - extracted tokens from figure 1.2 string

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)))
    ((eof) (token-EOF))
    (";" (token-SEMICOLON))
    ("pass" (token-PASS))
    ("break" (token-BREAK))
    ("continue" (token-CONTINUE))
    ("=" (token-ASSIGNMENT))
     ⋮
     ⋮
    ("[" (token-OB))
    ("]" (token-CB))
    ("True" (token-TRUE))
    ("False" (token-FALSE))
    ("None" (token-NONE))))
    ((:+ (:or (char-range #\0 #\9) (char-range #\a #\z) (char-range
#\A #\Z) #\_)) (token-ID lexeme))
```

Figure 2.4 – lexer function code to split the input string to its tokens

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 (:+)

2.2: how to design a parser?

After we've built the lexer, we can get our program tokens. Now it's the time to design a parser to build the program tree. Parser will build the program tree based on the tokens made by lexer. We use parser function defined in parser-tools/yacc library in racket.

So what is a program tree exactly?

Lets go back to our previous example (figure 2.2). we first should change that text in a way that our parser can parse this to a program tree. Remember that in our simplified version of python, we should put ';' after each statement – as well as function declarations, for loops, …) so we will change the text like that:

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

Figure 2.5 - edited version of our python code. this code can parse to a program tree

Now we want to parse this code. After doing that, we should get a program tree as below:

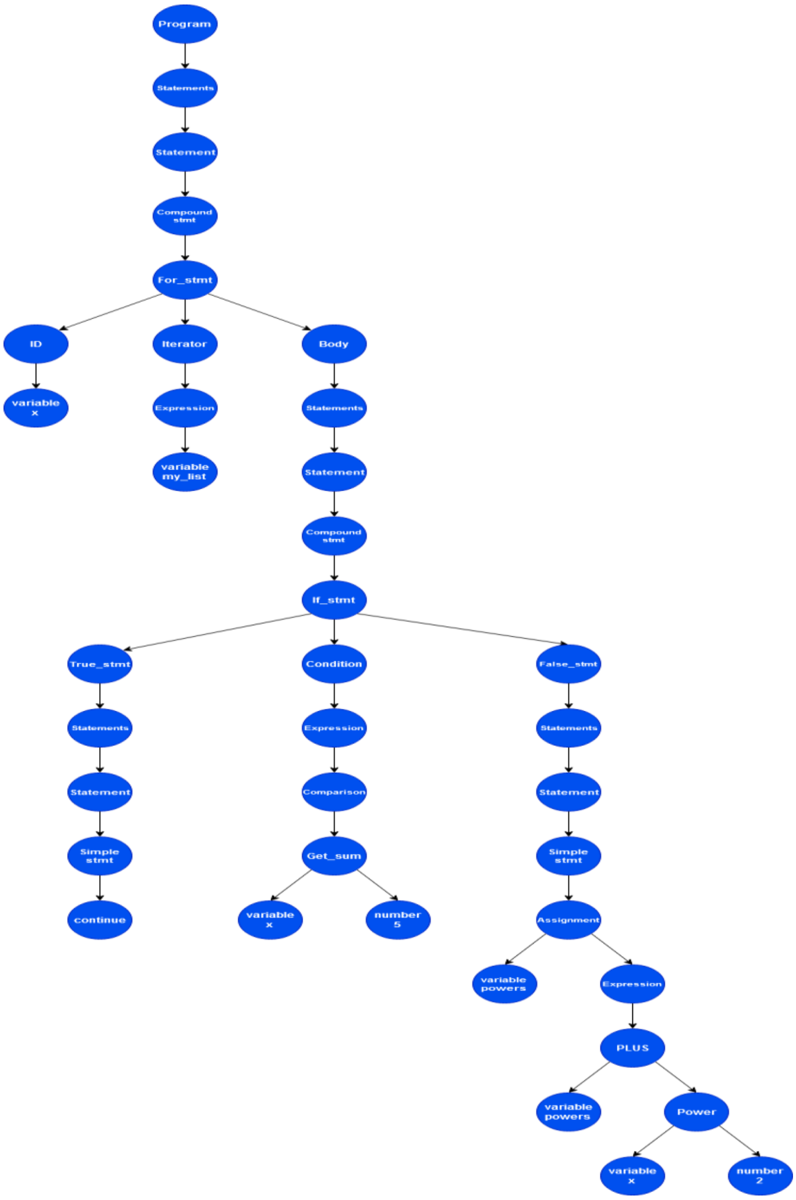

Figure 2.6 - parsed program tree for figure 2.5 code

So we write a parser function for our grammar. Notice that every line in our grammar represent a non-terminal-id. The following code shows the implementation of our parser:

```
(define full-parser
  (parser
   (start program)
   (end EOF)
   (error void)
   (tokens a b)
   (grammar
    (program ((statements) (list 'program $1)))
    (statements
     ((statement SEMICOLON) (list 'statements $1))
     ((statements statement SEMICOLON) (list 'statements $1 $2)))
    (statement
     ((comp-stmt) (list 'statement $1))
     ((smpl-stmt) (list 'statement $1)))
    (smpl-stmt
     ((assignment) (list 'smpl-stmt $1))
     ((glbl-stmt) (list 'smpl-stmt $1))
     ((rtrn-stmt) (list 'smpl-stmt $1))
     ((PASS) (list 'smpl-stmt 'pass))
     ((BREAK) (list 'smpl-stmt 'break))
     ((CONTINUE) (list 'smpl-stmt 'continue)))
    (comp-stmt
     ((func-def) (list 'comp-stmt $1))
     ((if-stmt) (list 'comp-stmt $1))
     ((for-stmt) (list 'comp-stmt $1)))
    (assignment ((ID ASSIGNMENT expression) (list 'assignment (list 'variable $1) $3)))
    (rtrn-stmt
     ((RETURN) (list 'return null))
     ((RETURN expression) (list 'return $2)))
    (glbl-stmt ((GLOBAL ID) (list 'global 'variable $2)))
    (func-def
     ((DEF ID OP CP COLON statements) (list 'function (list 'name $2) null (list 'body
$6)))
     ((DEF ID OP parameters CP COLON statements) (list 'function (list 'name $2) $4 (list
'body $7))))
    (parameters
     ((assignment) (list 'params $1))
     ((parameters COMMA assignment) (list 'params $1 $3)))
            ...
```

Figure 2.7 - the code of our implemented parser function