TenorC

Developed by @danii_mor

DEVELOPER MANUAL

LEXICAL ANALYSIS

First of all we start by creating our keywords. In them you can detail the tokens that you could not afford to use in any other way. They usually identify native language functions.

```
# tokenizing rules
    reserved = {
                'if'    : 'IF',
                'else'  : 'ELSE',
                'printf': 'PRINT',
                'switch': 'SWITCH',
                'goto'  : 'GOTO',
                'xor'   : 'XOR',
                #'sizeof': 'SIZE',
                'for'   : 'FOR',
                'case'  : 'CASE',
                'int'   : 'INT',
                'double': 'DOUBLE',
                'float' : 'FLOAT',
                'char'  : 'CHAR',
                'return': 'RETURN',
                'do'    : 'DO',
                'while' : 'WHILE',
                'struct': 'STRUCT',
                'break' : 'BREAK',
                'continue':'CONTINUE',
                'default': 'DEFAULT',
                'void'  : 'VOID',
                'scanf' : 'READ'
                }
```

To later declare the values with which our application will be interacting with. We need to do it separately because PLY will understand is just an ID of LABEL for then we can remark. That in effective that word ID or LABEL is a reserved word and need to be changed to its own type node.

```
tokens = [
                'ID',
                'NUMBER',
                'STRING'
            ] + list(reserved.values())

    literals = ['=', '\'', '"', '+', '-
', '*', '/', '%', '&', '|', '^', '<', '>', '!', '~', '(', ')', '[', ']', '{', '}'
, ';', ':', '.', ',', '?']

    t_ignore = " \t"

    t_ignore_COMMENT = r'[/] ( [/].* | [*].*[*][/] )'
```

For this we need to indicate what patterns this data will follow. Therefore, each token needs a function that defines the pattern to follow and some action that is required to be done just when it is found.

```
def t_STRING(t):
        r'( \"([^\\\n]|(\\.))*?\"  | \'([^\\\n]|(\\.))*?\' )'
        t.value = str(t.value).replace("\"","")
        t.value = str(t.value).replace("\'","")
        return t

    def t_NUMBER(t):
        r'\d+'
        t.value = int(t.value)
        return t

    def t_ID(t):
        r'[a-zA-Z][a-zA-Z_0-9]*'
        global sym_table
        # check if reserved word
        if t.value in reserved:
            t.type = reserved.get(t.value)
        #else:
            # add ID to symbol table
            #sym_table.add(str(t.value), 'ID', 0, None, 'GLOBAL')
            #sym_table.setScope(str(t.value))
        return t
            sym_table.setScope(str(t.value))
        return t
```

SYNTACTIC ANALYSIS

In this region we must indicate the patterns that our list of tokens must follow, already generated by our previous analysis.

```
s : code

code : code block
    | block

block : function
    |  struct
    |  declaration

function: type ID '(' argument_list  '{' compound_stament

type : INT
    | CHAR
    | DOUBLE
    | FLOAT

argument_list : arguments ')'
            | ')'

arguments : arguments ',' arg
        | arg

arg : type ID
    | type '&' ID

struct : STRUCT ID '{' assigment_list '}' ';'

assigment_list : assigment_list declaration ';'
            | declaration ';'

compound_stament : statement_list '}'
                | '}'

statement_list : statement_list statement
        | statement

statement : selection_statement
        | iteration_statement
        | declaration ';'
        | function_call ';'
        | jump_statement ';'
```

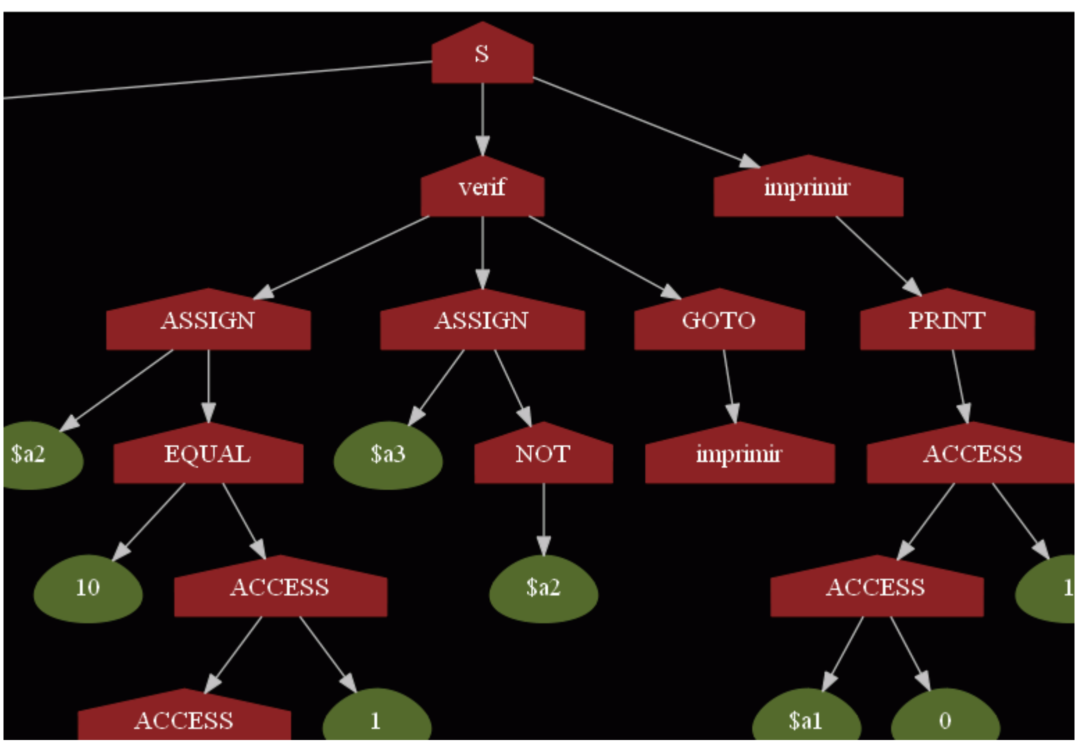
```
selection_statement : labeled_statement DEFAULT ':' statement_list "}"
                    | selection_if_has_more ELSE "{" compound_statement
                    | selection_if_has_more

selection_if_has_more : selection_if ELSE selection_if
                    |  selection_if

selection_if : IF "(" expression ")" "{" compound_statement

labeled_statement : labeled_statement CASE expression ':' statement_list
                | selection_switch

selection_switch : SWITCH "(" expression ")" "{"

iteration_statement : WHILE '(' expression ')' '{' compound_statement
                    | DO OCUR compound_statement WHILE '(' bool_expression ')' ';
'
                    | FOR '(' declaration ';' expression ';' unary_expr ')' '{'
                                                              compound_statement
jump_statement : CONTINUE
                | BREAK
                | RETURN expression
                | RETURN

declaration : type declaration_list
            | unary_expr

declaration_list : declaration_list ',' sub_decl
            | sub_decl

sub_decl : ID "=" expression
        | ID

assign_op : '='
        | '+' '='
        | '-' '='
        | '*' '='
        | '/' '='
        | '%' '='
        | '<' '<' '='
        | '>' '>' '='
        | '&' '='
        | '^' '='
        | '|' '='
```

```
unary_expr : '+' '+' is_array_term
        | '-' '-' is_array_term
        | is_array_term '+' '+'
        | is_array_term '-' '-'

is_array_term : is_array_term '[' term ']'
        | is_array_term '.' ID
        | ID

expression : expression '+' expression
        | expression '-' expression
        | expression '*' expression
        | expression '/' expression
        | expression '%' expression
        | expression '<' expression
        | expression '>' expression
        | expression '&' expression
        | expression '|' expression
        | expression '^' expression
        | expression XOR expression
        | '(' INT ')' expression
        | '(' FLOAT ')' expression
        | '(' CHAR ')' expression
        | '(' ID ')' expression
        | '(' expression ')'
        | '~' term
        | '!' expression
        | '-' expression
        | expression '?' expression ':' expression
        | ID '(' parentheses_expression
        | term
        | assigment_exp
        | expression '=' '=' expression
        | expression '!' '=' expression
        | expression '&' '&' expression
        | expression '|' '|' expression
        | expression '<' '=' expression
        | expression '>' '=' expression
        | expression '<' '<' expression
        | expression '>' '>' expression

term : STRING
    | NUMBER
    | NUMBER '.' NUMBER
    | is_array_term
```

it can be seen how a syntactic analysis tree is being generated among the grammatical productions. This will then help us to generate the orderly execution of our code.

```
def p_statement_list(p):
    '''list : list statement ";"
            | statement  ";" '''
    global sym_table
    if len(p) == 3:
        new_branch = branch()
        new_branch.add(p[1])
        p[0] = new_branch
        sym_table.appendGrammar(3, 'list -> statement ;')
    else:
        if p[1] != None:
            p[1].add(p[2])
            p[0] = p[1]
            sym_table.appendGrammar(4, 'list -> list statement ;')
        else:
            new_branch = branch()
            new_branch.add(p[2])
            p[0] = new_branch
            sym_table.appendGrammar(3, 'list -> statement ;')
```

Here you can clearly see how our tree is being generated as we go forward in our productions.



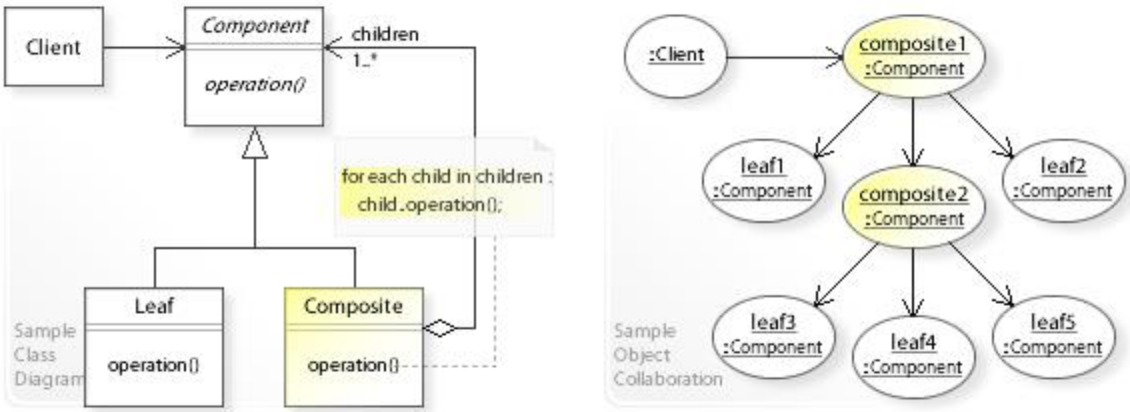
BACKPATCH

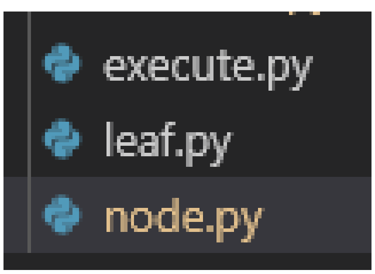
```python
def BACKPATCH(op, node, sym_table):
    e1 = node.getChild(0).execute(sym_table)
    e2 = node.getChild(1).execute(sym_table)

    # append generated code
    node.append3D(e1.get3D())

    # decide by op how to manage true and false labels
    if op == '&&':
        # print TRUE list of first argument
        node.gen3D('label', e1.getValue())

        # now TRUE list is from sencond argument
        node.setValue(e2.getValue())

        # append FALSE list of second argument
        node.setRef(e1.getRef() + ',' + e2.getRef())

    elif op == '||':
        # print FALSE list of first argument
        node.gen3D('label', e1.getRef())

        # append TRUE list of second argument
        node.setValue(e1.getValue() + ',' + e2.getValue())

        # now FALSE list of second argument
        node.setRef(e2.getRef())
    else:
        # print TRUE list of first argument
        node.gen3D('label', e1.getValue())

        # now TRUE list is FALSE list from sencond argument
        node.setValue(e2.getRef())

        # now FALSE list is TRUE list from sencond argument
        node.setRef(e2.getValue())

    node.append3D(e2.get3D())

    return node
```

HOW CAN I USE AST FROM C AND ALSO FOR AUGUS (INTERMEDIATE CODE)

I use composite pattern….

that’s why always I can use the same reference to it. Only specifying in my symbol table how to interpret each tree.






```python
from syntax_tree.node import node

class leaf(node):
    # _value and _type inherits from node class
    def __init__(self, value, typ):
        self._value = value
        self._type = node.TYPE[typ]
        self._ref = value

    def setValue(self, value):
        self._value = value

    def setType(self, typ):
        self._type = node.TYPE[typ]

    def getValue(self):
        return self._value
```

I use the compise NODE

For generate all characteristics that will be usefull in each type node. Leaf or Branch. And each one would have the same functions makings that I can call it recursively and not matter what kind of node it is.

HOW CAN I MAKE THE DEBUG OF INTERMEDIATE CODE WITH READ TERMINAL INPUT

I use a singleton pattern

So, wen I call again the new line to execute. It is a new hole process, lexical and syntaxis analysis. But the variables and changes made by the past lines we all have it in out Symbol Table.

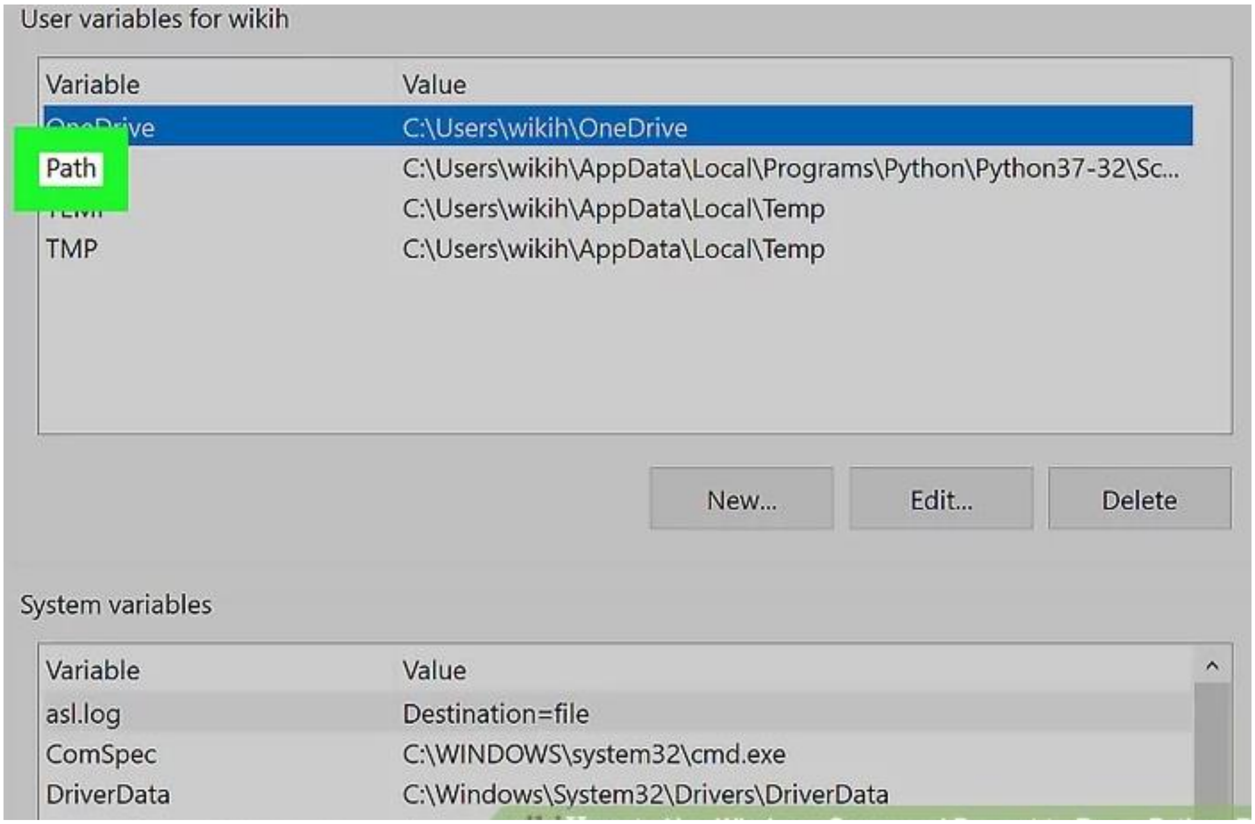
```
if self.__sym_table_3d != None:
     new_table =  {**self.__sym_table_3d.printTable(), **result[1].printTable()}
                        for sym_id in new_table:
                            sym = new_table[sym_id]
                            if sym != None:
                                if type(sym) == dict:
                                    continue
                                if sym.getValue() == None:
                                    try:
                                        new_table[sym_id] =
                                         self.__sym_table_3d.printTable()[sym_id]
                                    except:
                                        pass
  self.__sym_table_3d.setTable({**self.__sym_table_3d.printTable(), **new_table})
                    else:
                        self.__sym_table_3d = result[1]
                        # define mode for syntax-tree know how to autoexecute
                        self.__sym_table_3d.setMode(1)
```

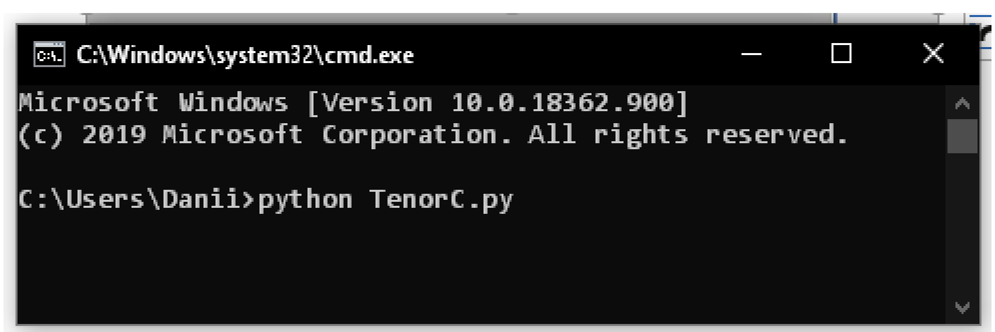
Now we can see how I merge the last symbol table changes with the new one making this only having one actual table in our global scope. Where we can use it.

For other purposes python give us to see that there are dictionaries that only uses the same theory of data bases. To copy all different keys in the new group.

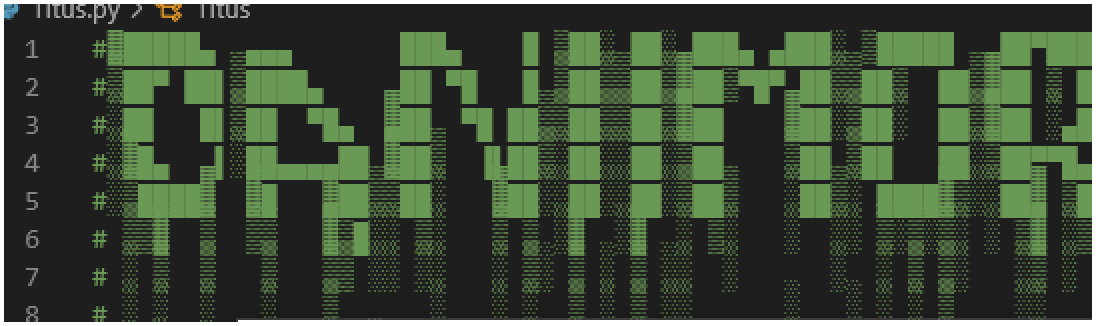
USAGE

We need Python3.8 + and added our installation of Graphviz / bin to our windows PATH. For linux we only need to update our Python + to the latest version and finally verify that the DOT compilation is accessed from BASH.

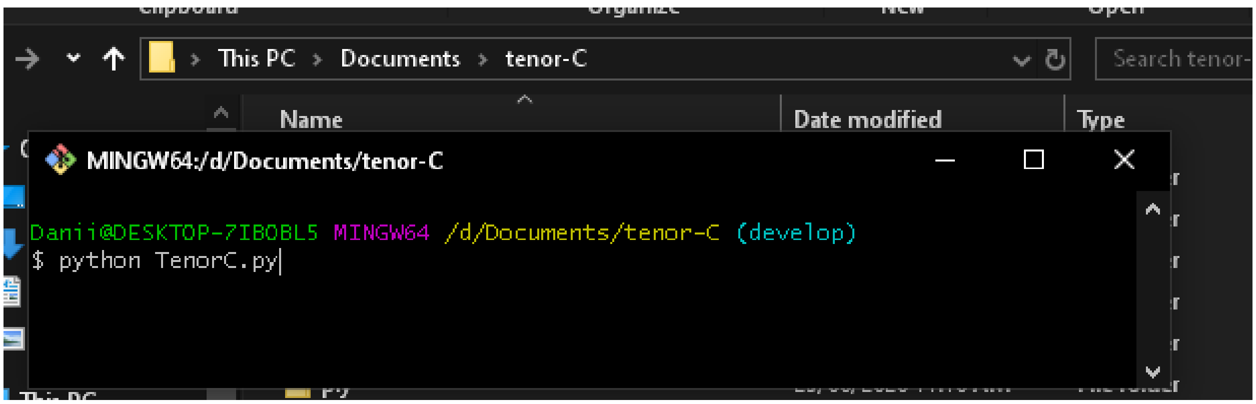




EXECUTE



Titus.py > Titus



This PC > Documents > tenor-C
Search tenor-
Name
Date modified
Type
MINGW64:/d/Documents/tenor-C
Danii@DESKTOP-7IBOBL5 MINGW64 /d/Documents/tenor-C (develop)
$ python TenorC.py