

PASCAL Code Compiler to Three Address Code

Compiler Construction



June 13, 2020

**SANA AZAM**

**SP17-BCS-014**

**BCS – 7A**

**Submitted to: Sir Salman Aslam**

**PROJECT REPORT**

Table of Contents

[1. Introduction 1](#_Toc42895275)

[2. Compiler’s Structure 1](#_Toc42895276)

[3. Programming Language used 2](#_Toc42895277)

[Code body 2](#_Toc42895278)

[Declaration 2](#_Toc42895279)

[Function 2](#_Toc42895280)

[Procedure 2](#_Toc42895281)

[Code block 2](#_Toc42895282)

[Conditional statements 2](#_Toc42895283)

[Loop statements 3](#_Toc42895284)

[4. Working 3](#_Toc42895285)

[Scanning 3](#_Toc42895286)

[Semantic Analysis 3](#_Toc42895287)

[Code Generation 3](#_Toc42895288)

[When Error 4](#_Toc42895289)

[5. Compiler’s Structure 5](#_Toc42895290)

[Identifiers 5](#_Toc42895291)

[Procedure Statements 5](#_Toc42895292)

[Simple Expressions 6](#_Toc42895293)

[Program Definition 7](#_Toc42895294)

[Variable Definition 7](#_Toc42895295)

[6. Compiler’s parsing Order 8](#_Toc42895296)

[7. Compiler’s Execution 8](#_Toc42895297)

[Code Input 8](#_Toc42895298)

[Three Address Code 9](#_Toc42895299)

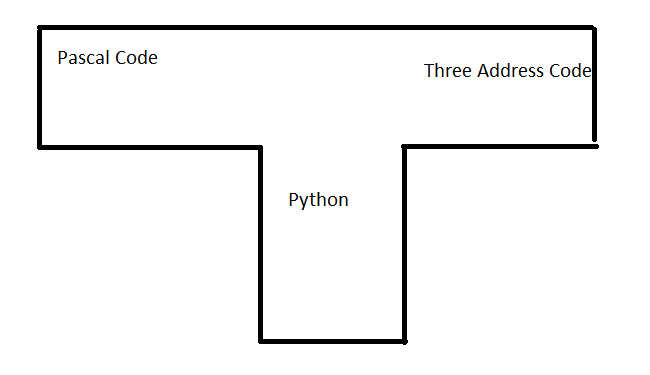
[Symbol Table 10](#_Toc42895300)

[8. Conclusion 10](#_Toc42895301)

# Introduction

As we know, compiler is a program that converts high-level language to assembly language. A compiler translates the code written in one language to some other language without changing the meaning of the program. Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

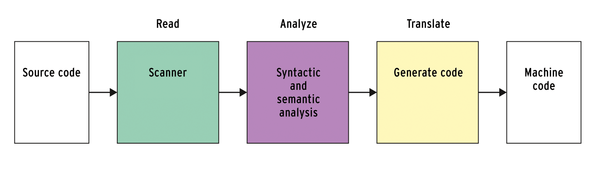
So, the compiler we designed is basically created on python where a Pascal code is translated into three-address code (TAC)



# Compiler’s Structure

It consists of three major parts which are:

1. Lexical Analyzer(Scanner)
2. Syntactic and semantic analyzer
3. Code Generation



# Programming Language used

We have used Pascal code to create our compiler as it is a very readable programming language. We can format your source code so that it reflects the flow of the program, and can use proper upper and lower case identifiers for variables and functions/procedures/methods.

# Code body

There are certain things which must be used exactly in that way to compile your code on this mini compiler.

## Declaration

“program example(input, output);

var i,j : integer;”

code is declared by var and by assigning other variables.

## Function

If your code contains a function, then it must be written as

*function mult(x:integer; y:integer):integer;*

*begin*

*mult := x \* y*

*end;*

## Procedure

If it contains a procedure, then it’s body must be like

*procedure toScreen(x:integer);*

*var y: integer;*

*begin*

*y := x;*

*writeln(y)*

*end;*

## Code block

Code must start with the “begin” keyword and will end with the “end”. A code may have conditional statements or sometimes a loop.

### Conditional statements

begin

i := 1;

j := -1;

if (i > 0) then

i := (1 - j\*3)

else

i := i + 1;

writeln(i)

end.

### Loop statements

*while (i<10) do begin*

*j := (i + j);*

*i := i + 1*

*end;*

# Working

### Scanning

Every input is read word by word by the compiler and later group these words by tokens.

### Semantic Analysis

Whenever an ID or a NUMBER is found, it checks that whether that input already exists or not. If not, then it simply adds its value in the table. Other than that with every assignment statement, there must be a variable or expression. If it doesn’t find any, it throws an error.



### Code Generation

In this compiler code generation occurs whenever a variable, expression, function or a function call is parsed.

### When Error

As, compiler reads one input at a time. Inputs are matched with the regular expressions. If, a certain input doesn’t match the regular expression, it will report an error.

        # To track error depth - we do not want to recover from sibling or

        # parent's functions.

        global depth

        # If we encounter an illegal symbol, skip passed it and report.

        if self.\_\_getTokenCode() == 'tc\_ERROR':

            message = 'Illegal character'

            while self.\_\_getTokenCode() == 'tc\_ERROR':

                self.\_\_addError(self.\_\_currentToken,message)

                self.\_\_getToken()

        # If we know there is an error, check depth and exit if depth is the same

        # or greater. Recover if error depth is smaller.

        if self.\_\_foundError:

            if self.\_\_errorDepth < depth:

                if self.\_\_getTokenCode() in syncsets[self.\_\_errorInFunction]:

                    if self.printTree: print '\t','    '\*depth,'--->Skipping', self.\_\_getTokenCode(), 'because of the Error flag.'

                    return

            else:

                #if self.printTree: print '\t','    '\*depth,self.\_\_errorInFunction, syncsets[self.\_\_errorInFunction]

                self.\_\_recover()

                self.\_\_getToken()

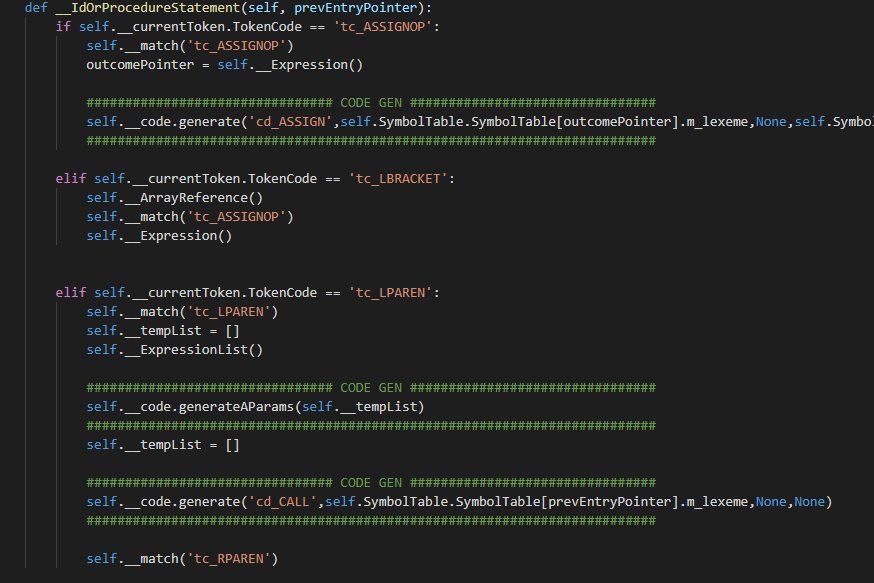
                return

# Compiler’s Structure

### Identifiers



### Procedure Statements



### Simple Expressions

def \_\_SimpleExpression(self,input=None):

        uminus = False

        if self.\_\_currentToken.TokenCode == 'tc\_ADDOP':

            op = self.\_\_currentToken.DataValue[1]

            if op == 'op\_MINUS':

                uminus = True

            self.\_\_match('tc\_ADDOP')

        entry = self.\_\_Term()

        entry = self.\_\_SimpleExpressionAddop(entry)

        if uminus:

            ################################ CODE GEN ################################

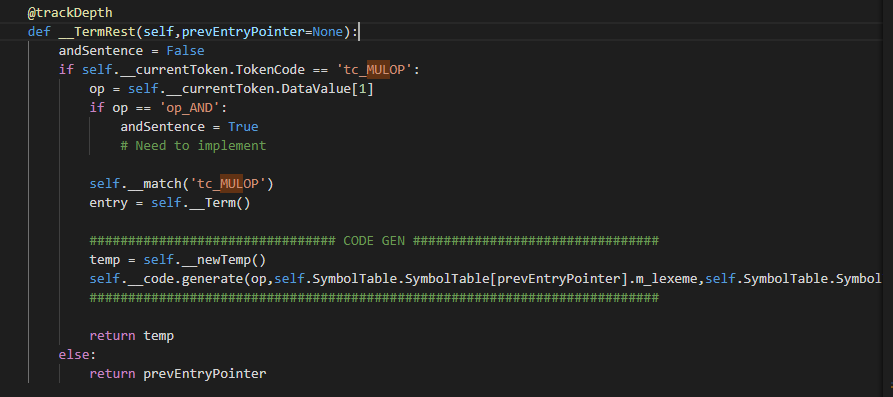
            temp = self.\_\_newTemp()

            self.\_\_code.generate('cd\_UMINUS',self.SymbolTable.SymbolTable[entry].m\_lexeme,None,self.SymbolTable.SymbolTable[temp].m\_lexeme)

            entry = temp

            ##########################################################################

        return entry



### Program Definition

def \_\_ProgramDefinition(self,input=None):

        self.\_\_match('tc\_PROGRAM')

        pointer = self.\_\_currentToken.getSymTabEntry()

        name = self.SymbolTable.SymbolTable[pointer].m\_lexeme

        self.\_\_match('tc\_ID')

        self.\_\_match('tc\_LPAREN')

        self.\_\_IdentifierList()

        self.\_\_match('tc\_RPAREN')

        return name

### 

### Variable Definition

 @trackDepth

    def \_\_IdentifierList(self,input=None):

        lexeme = self.SymbolTable.SymbolTable[self.\_\_currentToken.getSymTabEntry()].m\_lexeme

        self.\_\_tempList.append(lexeme)

        self.\_\_match('tc\_ID')

        if self.\_\_currentToken.TokenCode == 'tc\_COMMA': self.\_\_IdentifierListRest()

    @trackDepth

    def \_\_IdentifierListRest(self,input=None):

        self.\_\_match('tc\_COMMA')

        self.\_\_IdentifierList()

    @trackDepth

    def \_\_IdentifierListAndType(self,input=None):

        self.\_\_IdentifierList()

        self.\_\_match('tc\_COLON')

        self.\_\_Type()

# Compiler’s parsing Order

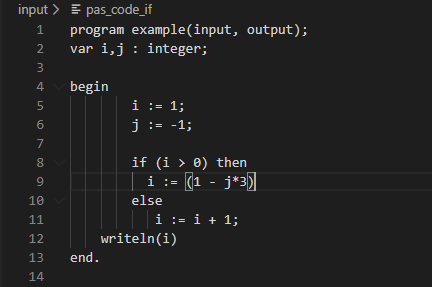
It parses the following in order

1. Program definition
2. Variable declaration
3. Function or Procedures declaration
4. Main program must start with “begin” and terminate with “end.”

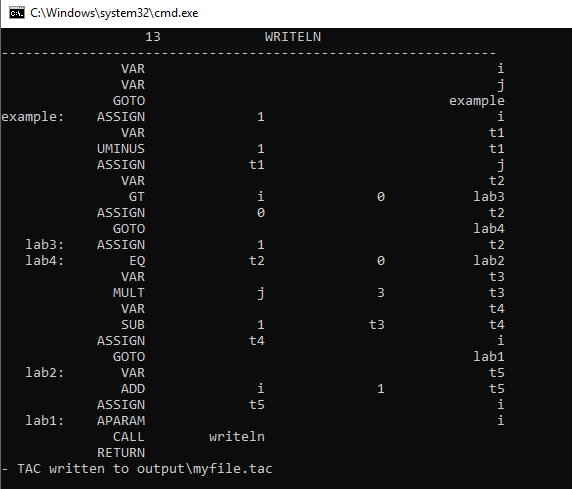
# Compiler’s Execution

After executing, it will show you the Three Address Code(TAC) and symbol table.

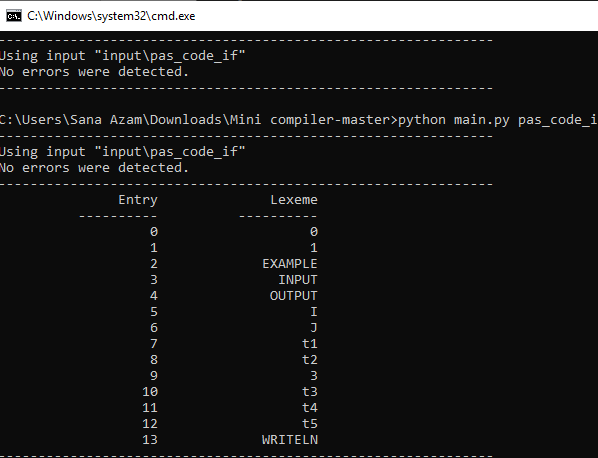
### Code Input



### Three Address Code



### Symbol Table



# Conclusion

Our mini compiler covers all the major components of a compiler that are lexical analyzer, parser, semantic analyzer and code generation. It defines some basic possibilities that how a compiler might work for a high level language. It is a great way to reduce system load, improve performance and productivity.