TOY COMPILER

## A PROJECT REPORT

***Submitted by***

**P. SAI MANOJ(19113101)**

**K. SUMANTH KUMAR REDDY(19113105)**

**P. UMESH CHANDRA(19113118)**

**Under the guidance of**

### Dr.SATHYAPRIYA.S

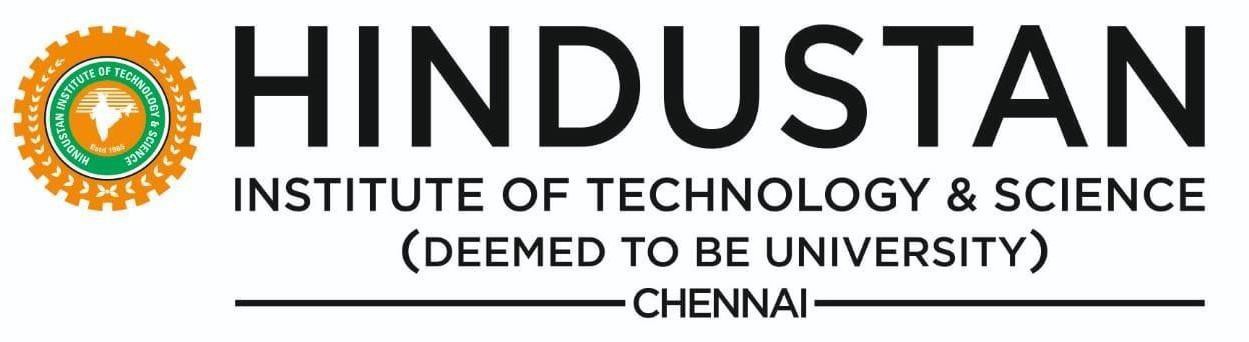
***In partial fulfillment for the award of the degree of***

## BACHELOR OF TECHNOLOGY

***In***

## COMPUTER SCIENCE AND ENGINEERING

## 



# **BONAFIDE CERTIFICATE**

Certified that this project report **“TOY COMPILER”** is the bonafide work of “**P. SAI MANOJ (19113101), K. SUMANTH KUMAR REDDY (19113105), and P. UMESH CHANDRA (19113118)”** who carried out the project work under my supervision during the academic year 2**021-2022**.

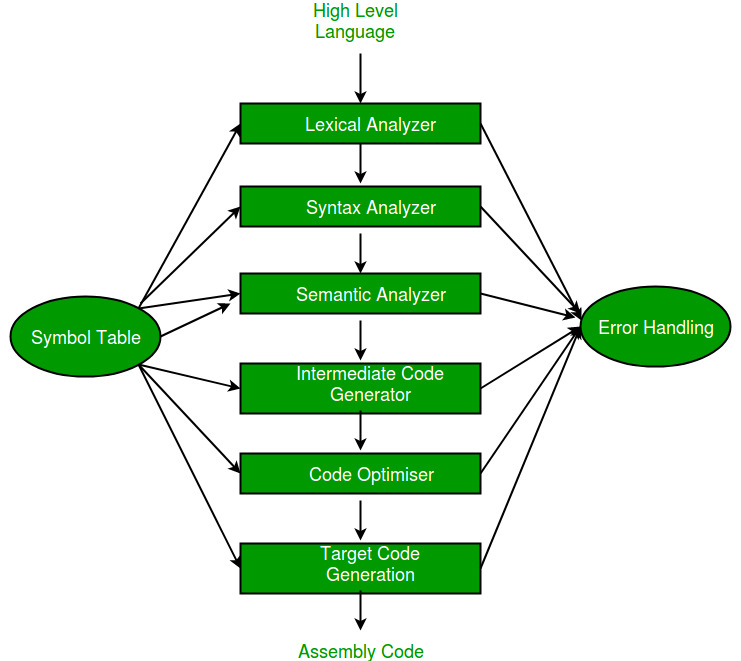
**Dr S SATYA PRIYA**

**ABSTRACT**

Toy compiler (informal) specification Lexer Parser Code generation Coding tips Submitting your project via GitHub (Live coding) Readings Wrap-up Resources Makefile Overview A compiler is a language translator Bridges human-friendlier languages and machine code Toy compiler for arithmetic expressions Take binary arithmetic operations.

**INTRODUCTION**

We basically have two phases of compilers, namely the Analysis phase and Synthesis phase. The analysis phase creates an intermediate representation from the given source code. The synthesis phase creates an equivalent target program from the intermediate representation.



Symbol Table – It is a data structure being used and maintained by the compiler, consisting of all the identifier’s names along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

The analysis of a source program is divided into mainly three phases. They are:

Linear Analysis-

This involves a scanning phase where the stream of characters is read from left to right. It is then grouped into various tokens having a collective meaning.

Hierarchical Analysis-

In this analysis phase, based on a collective meaning, the tokens are categorized hierarchically into nested groups.

Semantic Analysis-

This phase is used to check whether the components of the source program are meaningful or not.

The compiler has two modules namely the front end and the back end. Front-end constitutes the Lexical analyzer, semantic analyzer, syntax analyzer, and intermediate code generator. And the rest are assembled to form the back end.

Lexical Analyzer –

It is also called a scanner. It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure high-level language. It reads the characters from the source program and groups them into lexemes (sequence of characters that “go together”). Each lexeme corresponds to a token. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes lexical errors (e.g., erroneous characters), comments, and white space.

Syntax Analyzer – It is sometimes called a parser. It constructs the parse tree. It takes all the tokens one by one and uses Context-Free Grammar to construct the parse tree.

Why Grammar?

The rules of programming can be entirely represented in a few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.

The parse tree is also called the derivation tree. Parse trees are generally constructed to check for ambiguity in the given grammar. There are certain rules associated with the derivation tree.

Any identifier is an expression

Any number can be called an expression

Performing any operations in the given expression will always result in an expression. For example, the sum of two expressions is also an expression.

The parse tree can be compressed to form a syntax tree

Syntax error can be detected at this level if the input is not in accordance with the grammar.

Semantic Analyzer – It verifies the parse tree, whether it’s meaningful or not. It furthermore produces a verified parse tree. It also does type checking, Label checking, and Flow control checking.

Intermediate Code Generator – It generates intermediate code, which is a form that can be readily executed by a machine We have many popular intermediate codes. Example – Three address codes etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.

Till intermediate code, it is the same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don’t need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

Code Optimizer – It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered. Optimization can be categorized into two types: machine-dependent and machine-independent.

Target Code Generator – The main purpose of the Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection, etc. The output is dependent on the type of assembler. This is the final stage of compilation. The optimized code is converted into relocatable machine code which then forms the input to the linker and loader.

**Source Code:**

analysisphase.py

import re

import nltk

import ast

import astpretty

from subprocess import call

OPERATORS = set(['+', '-', '\*', '/', '(', ')'])

PRI = {'+':1, '-':1, '\*':2, '/':2}

def lex\_code(exp):

input\_program = exp

input\_program\_tokens = nltk.wordpunct\_tokenize(input\_program);

print("\n\n \t\t\t\t\\*lexical analyzer\*")

print("\n Tokens: \n {}".format(input\_program\_tokens));

RE\_Keywords = "auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|long|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|while|string|class|struc|include"

RE\_Operators = "(\++)|(-)|(=)|(\\*)|(/)|(%)|(--)|(<=)|(>=)"

RE\_Numerals = "^(\d+)$"

RE\_Special\_Characters = "[\[@&~!#$\^\|{}\]:;<>?,\.']|\(\)|\(|\)|{}|\[\]|\""

RE\_Identifiers = "^[a-zA-Z\_]+[a-zA-Z0-9\_]\*"

RE\_Headers = "([a-zA-Z]+\.[h])"

#To Categorize The Tokens

for token in input\_program\_tokens:

if(re.findall(RE\_Keywords,token)):

print(token , "-------> Keyword")

elif(re.findall(RE\_Operators,token)):

print(token, "-------> Operator")

elif(re.findall(RE\_Numerals,token)):

print(token, "-------> Numeral")

elif(re.findall(RE\_Special\_Characters,token)):

print(token, "-------> Special Character/Symbol")

elif(re.findall(RE\_Identifiers,token)):

print(token, "-------> Identifiers")

else:

print("Unknown Value")

def syntax\_analysis(exp2):

print("\n\n \t\t\t\t\\*syntax analyzer\*")

output=astpretty.pprint(ast.parse(exp2).body[0])

print(output)

### INFIX ===> POSTFIX ###

def infix\_to\_postfix(formula):

stack = [] # only pop when the coming op has priority

output = ''

for ch in formula:

if ch not in OPERATORS:

output += ch

elif ch == '(':

stack.append('(')

elif ch == ')':

while stack and stack[-1] != '(':

output += stack.pop()

stack.pop() # pop '('

else:

while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:

output += stack.pop()

stack.append(ch)

# leftover

while stack:

output += stack.pop()

print(f'POSTFIX: {output}')

return output

### INFIX ===> PREFIX ###

def infix\_to\_prefix(formula):

print("\n\n\t\t\t\*\* Intermediate code generation \*\*\*")

op\_stack = []

exp\_stack = []

for ch in formula:

if not ch in OPERATORS:

exp\_stack.append(ch)

elif ch == '(':

op\_stack.append(ch)

elif ch == ')':

while op\_stack[-1] != '(':

op = op\_stack.pop()

a = exp\_stack.pop()

b = exp\_stack.pop()

exp\_stack.append( op+b+a )

op\_stack.pop() # pop '('

else:

while op\_stack and op\_stack[-1] != '(' and PRI[ch] <= PRI[op\_stack[-1]]:

op = op\_stack.pop()

a = exp\_stack.pop()

b = exp\_stack.pop()

exp\_stack.append( op+b+a )

op\_stack.append(ch)

# leftover

while op\_stack:

op = op\_stack.pop()

a = exp\_stack.pop()

b = exp\_stack.pop()

exp\_stack.append( op+b+a )

print(f'PREFIX: {exp\_stack[-1]}')

return exp\_stack[-1]

### THREE ADDRESS CODE GENERATION ###

def generate3AC(pos):

print("### THREE ADDRESS CODE GENERATION ###")

exp\_stack = []

t = 1

for i in pos:

if i not in OPERATORS:

exp\_stack.append(i)

else:

print(f't{t} := {exp\_stack[-2]} {i} {exp\_stack[-1]}')

exp\_stack=exp\_stack[:-2]

exp\_stack.append(f't{t}')

t+=1

def opencodegenerator():

call(["python", "codegenerator.py"])

expres = input("INPUT THE EXPRESSION: ")

exp = lex\_code(expres)

exp2 = syntax\_analysis(expres)

pre = infix\_to\_prefix(expres)

pos = infix\_to\_postfix(expres)

generate3AC(pos)

opencodegenerator()

codegenerator.py

import re

import nltk

import ast

import astpretty

from subprocess import call

OPERATORS = set(['+', '-', '\*', '/', '(', ')'])

PRI = {'+':1, '-':1, '\*':2, '/':2}

def lex\_code(exp):

input\_program = exp

input\_program\_tokens = nltk.wordpunct\_tokenize(input\_program);

print("\n\n \t\t\t\t\\*lexical analyzer\*")

print("\n Tokens: \n {}".format(input\_program\_tokens));

RE\_Keywords = "auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|long|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|while|string|class|struc|include"

RE\_Operators = "(\++)|(-)|(=)|(\\*)|(/)|(%)|(--)|(<=)|(>=)"

RE\_Numerals = "^(\d+)$"

RE\_Special\_Characters = "[\[@&~!#$\^\|{}\]:;<>…

[1:23 PM, 4/20/2022] Manoj: import re

f = open("intermediate.txt", 'r')

lines = f.readlines()

f.close()

print("\n\n\t\t\t\*\* code generation \*\*\*")

fifo\_return\_reg = 'R0'

reg=[0]\*13

var={} # {'a': 'R0', '1':'R0'}

store\_seq=[]

fifo\_reg = 0

operator\_list = {'+':'ADD', '-':'SUB', '\*':'MUL', '/':'DIV', '==':'E', '!=':'NE', '>':'G', '<':'L', '>=':'GE','<=':'LE','and':'AND','or':'OR'}

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

def fifo():

#print(" In FIFO Function", var)

#print("\n\n Store seq", store\_seq)

global fifo\_reg

global fifo\_return\_reg

for k,v in var.copy().items():

if(v == 'R'+str(fifo\_reg)):

fifo\_return\_reg = v

# print("K", k)

# print("V", v)

var.pop(k)

if(k in store\_seq):

store\_seq.remove(k)

print("ST ", k, ', ', v, sep='')

fifo\_reg = int(fifo\_return\_reg[1:]) + 1

# print("FIFO reg & v", fifo\_reg, fifo\_return\_reg)

return fifo\_return\_reg

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

def getreg():

for i in range(0,13):

if reg[i]==0:

reg[i]=1

return 'R'+str(i)

register = fifo()

# print("REGISTER" , register)

return register

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

for line in lines:

line = line.strip()

line = line.strip('\n')

line = line.split()

length = len(line)

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

#branch

#L0:

if(length == 1):

print(line[0])

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

# temporary variables

# ignore statement

# re.findall('^t[0-9]+$')

elif(re.findall('^t[0-9]+$', line[0])):

continue

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

# goto

#simple assignment

#a = b

#LD Rn, b

elif(length == 3):

lhs = line[0]

operand = line[2]

if operand not in var:

var[operand] = getreg()

if(operand.isalpha()):

print("LD ", var[operand], ', ', operand, sep ='')

else:

print("MOV ", var[operand], ', #', operand, sep ='')

#remove the old occurence and put new one -->

if lhs in store\_seq:

old\_reg = var[lhs]

index = store\_seq.index(lhs)

store\_seq.pop(index)

var[lhs] = var[operand]

if(old\_reg not in var.values()):

reg[int(old\_reg[1:])] = 0

else:

var[lhs] = var[operand]

store\_seq.append(lhs)

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

elif('goto' in line):

# if x goto l

if('if' in line):

operand = line[1]

label = line[3]

if operand not in var:

var[operand] = getreg()

if(operand.isalpha()):

print("LD ", var[operand], ', ', operand, sep ='')

else:

print("MOV ", var[operand], ', #', operand, sep ='')

print("BNEZ ", var[operand], ', ', label, sep='')

# goto l

else:

print("BR",line[1])

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

# assignment expressions

# +, -, \*, /, >, <, <=, >=, ==, !=, not, and, or

#a = b + c

else:

if(len(line)==5):

oper = line[3]

operand1 = line[2]

operand2 = line[4]

lhs = line[0]

if operand1 not in var:

var[operand1] = getreg()

if(operand1.isalpha()):

print("LD ", var[operand1], ', ', operand1, sep ='')

else:

print("MOV ", var[operand1], ', #', operand1, sep ='')

if operand2 not in var:

var[operand2] = getreg()

if(operand2.isalpha()):

print("LD ", var[operand2], ', ', operand2, sep ='')

else:

print("MOV ", var[operand2], ', #', operand2, sep ='')

operator\_print = operator\_list[oper]

#remove the old occurence and put new one -->

if lhs in store\_seq:

old\_reg = var[lhs]

index = store\_seq.index(lhs)

store\_seq.pop(index)

var[lhs] = getreg()

reg[int(old\_reg[1:])] = 0

else:

var[lhs] = getreg()

store\_seq.append(lhs)

print(operator\_print, ' ', var[lhs], ', ', var[operand1], ', ', var[operand2], sep='')

else:

operand = line[3]

lhs = line[0]

if operand not in var:

var[operand] = getreg()

if(operand.isalpha()):

print("LD ", var[operand], ', ', operand, sep ='')

else:

print("MOV ", var[operand], ', #', operand, sep ='')

if lhs in store\_seq:

old\_reg = var[lhs]

index = store\_seq.index(lhs)

store\_seq.pop(index)

var[lhs] = getreg()

reg[int(old\_reg[1:])] = 0

else:

var[lhs] = getreg()

store\_seq.append(lhs)

print("NOT ", var[lhs], ', ', var[operand], sep='')

# store all variables

for i in store\_seq:

print("ST ", i, ', ', var[i], sep='')

Outputs

