**Building a Compiler of my own**

**There are 4 building blocks**

1. Taking input code and parsing it to a list of tokens.
2. Convert that list of tokens to a correct assembly program.
3. Using the **nasm assembler** in order to assemble our assembly program files into an object file.
4. Use a **linker** such as **GCC** to link and our final executable.

**program0.oll**

PUSH 10

PUSH 7

ADD

PRINT "Hello, World"

HALT

**program1.oll**

READ

READ

SUB

JUMP.EQ.0 L1

PRINT "not equal"

HALT

L1:

PRINT "equal"

HALT

**program2.oll**

READ

JUMP.EQ.0 L1

L2:

PUSH 2

SUB

JUMP.EQ.0 L1

JUMP.GT.0 L2

PRINT "odd"

HALT

L1:

PRINT "even"

HALT

**program3.oll**

READ

JUMP.EQ.0 L2

JUMP.GT.0 L0

PRINT "Negative"

HALT

L0:

PUSH 3

SUB

JUMP.EQ.0 L1

JUMP.GT.0 L0

PRINT "Not Divisible by 3"

HALT

L1:

PRINT "Divisible by 3"

HALT

L2:

PRINT "Zero"

HALT

**Now… Step 1:**

1. Taking input code and parsing it to a list of tokens.

**compiler.py**

'''

Compiler for Out Little Language (.oll)

'''

import sys

#read arguments

program\_filepath = sys.argv[1]

print("[CMD] Parsing")

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

#     Tokenize Program

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

#read file lines

program\_lines = []

with open(program\_filepath, "r") as program\_file:

    program\_lines = [

        line.strip()

            for line in program\_file.readlines()]

program = []

for line in program\_lines:

    parts = line.split(" ")

    opcode = parts[0]

    # check for empty line

    if opcode == "":

        continue

    # store opcode token

    program.append(opcode)

    # handle each opcode

    if opcode == "PUSH":

        # expecting a number

        number = int(parts[1])

        program.append(number)

    elif opcode == "PRINT":

        # parse string literal

        string\_literal = ' '.join(parts[1:])[1:-1]

        program.append(string\_literal)

    elif opcode == "JUMP.EQ.0":

        # read label

        label = parts[1]

        program.append(label)

    elif opcode == "JUMP.GT.0":

        # read label

        label = parts[1]

        program.append(label)

print(program)

**Output:**

|  |
| --- |
|  |

**compile.py**

'''

Compiler for Out Little Language (.oll)

'''

import sys

#read arguments

program\_filepath = sys.argv[1]

print("[CMD] Parsing")

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

#     Tokenize Program

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

#read file lines

program\_lines = []

with open(program\_filepath, "r") as program\_file:

    program\_lines = [

        line.strip()

            for line in program\_file.readlines()]

program = []

for line in program\_lines:

    parts = line.split(" ")

    opcode = parts[0]

    # check for empty line

    if opcode == "":

        continue

    # store opcode token

    program.append(opcode)

    # handle each opcode

    if opcode == "PUSH":

        # expecting a number

        number = int(parts[1])

        program.append(number)

    elif opcode == "PRINT":

        # parse string literal

        string\_literal = ' '.join(parts[1:])[1:-1]

        program.append(string\_literal)

    elif opcode == "JUMP.EQ.0":

        # read label

        label = parts[1]

        program.append(label)

    elif opcode == "JUMP.GT.0":

        # read label

        label = parts[1]

        program.append(label)

'''

Compile to assembly

'''

asm\_filepath = program\_filepath[:-4] + ".asm"

out = open(asm\_filepath, "w")

out.write("""; -- header --

          bits 64

          default rel

          """)

out.write("""; -- variables --

          section .bss

          """)

out.write("""; -- contants --

          section .date

          """)

out.write("""; -- Entry Point --

          section .text

          global main

          extern ExitProcess

          extern printf

          extern scanf

          main:

          \tPUSH rbp

          \tMOV rbp, rsp

          \tSUB rsp, 32

          """)

ip = 0

while ip < len(program):

    opcode = program[ip]

    ip += 1

    if opcode.endswith(":"):

        out.write(f"; -- Label ---\n")

        out.write(f"{opcode}\n")

    elif opcode == "PUSH":

        number = program[ip]

        ip += 1

        out.write(f"; -- PUSH ---\n")

        out.write(f"\tPUSH {number}\n")

    elif opcode == "POP":

        out.write(f"; -- POP ---\n")

        out.write(f"\tPOP\n")

    elif opcode == "ADD":

        out.write(f"; --ADD ---\n")

        out.write(f"\tPOP rax\n")

        out.write(f"\tADD qword [rsp], rax\n")

    elif opcode == "SUB":

        out.write(f"; -- SUB ---\n")

        out.write(f"\tPOP rax\n")

        out.write(f"SUB qword [rsp], rax\n")

    elif opcode == "PRINT":

        string\_literal\_index = program[ip]

        ip += 1

        out.write(f"; -- PRINT ---\n")

        out.write(f"; NOT IMPLEMENTED \n")

    elif opcode == "READ":

        out.write(f"; -- READ ---\n")

        out.write(f"; NOT IMPLEMENTED \n")

    elif opcode == "JUMP.EQ.0":

        label = program[ip]

        ip += 1

        out.write(f"; -- JUMP.EQ.0 ---\n")

        out.write(f"\tCMP qword [rsp], 0\n")

        out.write(f"\tJE {label}\n")

    elif opcode == "JUMP.GT.0":

        label = program[ip]

        ip += 1

        out.write(f"; --JUMP.GT.0 ---\n")

        out.write(f"\tCMP qword [rsp], 0\n")

        out.write(f"\tJG {label}\n")

    elif opcode == "HALT":

        out.write(f"; --HALT ---\n")

        out.write(f"\tJMP EXIT\_LABEL\n")

out.write("EXIT\_LABEL:\n")

out.write(f"\tXOR rax, rax\n")

out.write(f"\tCALL ExitProcess\n")

out.close()

**Output:**

|  |
| --- |
|  |