Toy C Compiler

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY, ALLAHABAD

Language Used: Python

Members:

1. Roshan Baghwar IIT2017147
2. Akash Singh BIM2017004
3. Yashwant Panchal BIM2017006
4. Mayank Taskande ITM2017008
5. Ashutosh Kumar ISM2017003
6. Anup Bediya ISM2017002

Our Toy C Compiler is based on following stages;

Read in a C file then:

1. Lex the file
2. Parse the file
3. Semantic Analysis
4. Generate the assembly code of the file

☑ **Task 1:**

Write a program that accepts a C source file.

/* Main *(myCompiler.py)* */

```
import sys
import subprocess
import argparse

from lexer import *
import tokens
from parser import *
import rules
from code_gen import *

if __name__=="__main__":

   # Parse the command-line arguments
   parser = argparse.ArgumentParser(description='myCompiler')
```

```
    # The input .c file name
    parser.add_argument('input', metavar='cFile',
                        type=argparse.FileType('r'), help="the input c
file")

    # The output file name
    parser.add_argument('-o', metavar='outputFile', dest='output',
                        help="the name of the output file")

    # A flag to create only the asm file
    parser.add_argument('-S', dest='asm_only', action='store_const',
const=True,
                        default=False, help="create only the assembly
file")
    args = parser.parse_args()

    try:
        # Read the input file
        program_text = args.input.read()
    except:
        print("Could not read input file.")
    else:
        # If the file opened and was read, then carry on with tokenizing
        try:
            token_list = tokenize(program_text, tokens.prims)
        except TokenException as e: # catch any exceptions from the lexer
            print(e)
            sys.exit(1)
        else:
            try:
                # Parse the input into a syntax tree. See parser.py for
                # documentation on what all these parameters are.
                parse_root = generate_tree(token_list, rules.rules,
rules.S,
                                           tokens.comment_start,
                                           tokens.comment_end,
```

```
                                          add_rule = rules.E_add,
                                          neg_rule = rules.E_neg,
                                          mult_rule = rules.E_mult,
                                          pointer_rule = rules.E_point,
                                          dec_sep_rule =
rules.declare_separator_base,
                                          dec_exp_symbol =
rules.declare_expression)
           except ParseException as e: # catch any exceptions from the
parser
               print(e)
               sys.exit(1)
           else:
               try:
                   # As defined/explained in code_gen_obj.py
                   code = CodeManager()
                   info = StateInfo()

                   # Traverse the tree and generate asm into the
CodeManager object
                   info = make_code(parse_root, info, code)

                   # Check if main function exists and has right type
                   mainfunc = info.get_func("main")
                   if mainfunc["args"] or mainfunc["ftype"] != Type("int",
0): raise NoMainFunctionException()

                   # Saves code string to complete_code
                   complete_code = code.get_code(mainfunc["label"])
               except (RuleGenException,
                       VariableRedeclarationException,
                       VariableNotDeclaredException,
                       NoMainFunctionException) as e:
                   # Catch any exceptions from the code generation step

                   print(e)
                   sys.exit(1)
```

```
            else:
                try:
                    # Open the file for saving generated asm code
                    if args.output: output_name = args.output
                    else: output_name = args.input.name.split(".")[0]

                    g = open(output_name + ".s", "w")
                except:
                    print("Could not create output asm file.")
                else:
                    # Write the code to the file

                    g.write(complete_code)
                    g.close()

                    print("Compilation completed.")

                    # Compile the file into a final executable
                    # TODO: check version of nasm first
                    if not args.asm_only:
                        subprocess.call(["nasm", "-f", "macho64",
output_name + ".s"])
                        subprocess.call(["ld", output_name + ".o",
"-o", output_name])
                        print("Done.")
    finally:
        args.input.close()
```

/* Main Code ends */

☑ **Task 2:**

Write a LEXER program that accepts a C file and returns a list of tokens.

/* Lexer Code *(lexer.py)* */

```
import re

class Token:

   def __init__(self, name, text = None, priority = None):
       # the general type of token (e.g. "assignment" or "bracket")
       self.name = name

       # usually the literal text of the token (e.g. "=" or "{")
       self.text = text

       # the parser does not apply a rule if the next token is higher
priority
       # See the parser for more info.
       self.priority = priority

   def match(self, token):
       if not isinstance(token, Token): return False
       if self.name != token.name: return False
       if not self.text: return True
       return (self.text == token.text)
   def __eq__(self, other):
       """Checks if two tokens are exactly equal in name and text"""
       return ((self.name == other.name) and (self.text == other.text))
   def __repr__(self):
       return str(str(self.name) + " " + str(self.text))
```

```python
    def display(self, level = 0):
        """Debugging tool used to display tokens in a parse tree"""
        print("|    " * level + str(self))
    def bracket_repr(self):
        return self.text

class TokenException(Exception):
    """Exception thrown when a program cannot be split into tokens"""
    def __init__(self, bad_part):
        self.bad_part = bad_part
    def __str__(self):
        return "Error tokenizing part: " + self.bad_part

def tokenize(program, prims):
    """Executes the task of the lexer.
    program - the input program text, as a string
    prims - a list of the primitive tokens to split program into
    Returns a list of tokens representing the program.
    """

    # `parts` stores the current tokenization of the program
    # To begin, split the program by whitespace
    parts = re.split("\s+", program)

    # split by each of the primitives, in order
    for prim in prims:
        new_parts = [] # temporary storage for the next iteration of parts

        # for every part in current tokenization
        for part in parts:
            # if it isn't already a token, try splitting it up
            if isinstance(part, str):
                split = part.split(prim.text)

                # add results of spliting to new_parts
                for s in split:
                    if len(s) > 0: new_parts.append(s)
```

```
                    new_parts.append(prim)

                # we add one too many `prim`s above
                new_parts.pop()

            # if it's already a token, don't do anything
            else:
                new_parts.append(part)

        parts = new_parts

    # For each remaining string element of parts, tokenize it properly
    def make_token(part):
        # if it's already a token, don't change anything
        if isinstance(part, Token):
            return part
        # if it's a number, store it as an integer
        elif re.fullmatch("[0-9]*", part):
            return Token("integer", part)
        # if it's a valid name, stori it as a name
        elif re.fullmatch("[a-zA-Z_][a-zA-Z0-9_]*", part):
            return Token("name", part)
        else: # we've found something unexpected! complain.
            raise TokenException(part)

    return list(map(make_token, parts))
```

/* Lexer Code ends */

☑ **Task 3:**

Write a PARSER program to transform the list of tokens into the syntax tree.

/* Parser Code *(parser.py)* */

```
class ParseNode:

   def __init__(self, rule, children):
       self.rule = rule
       self.children = children
   def __repr__(self):
       return str(self.rule.orig) + " -> " + str(self.children)
   def display(self, level = 0):
       """Used for printing out the tree"""
       print("|    " * level + str(self.rule.orig))
       for child in self.children:
           child.display(level+1)
   def bracket_repr(self):
       outstr = "[" + str(self.rule.orig) + " "
       outstr += ' '.join(child.bracket_repr() for child in self.children)
       outstr += "]"
       return outstr

class ParseException(Exception):
```

```python
    def __init__(self, stack):
        self.stack = stack
    def __str__(self):
        return "Error parsing input.\nEnd tree: " + str(self.stack)

def generate_tree(tokens, rules, start_symbol,
                  comment_start, comment_end,
                  add_rule, neg_rule,
                  mult_rule, pointer_rule,
                  dec_sep_rule, dec_exp_symbol):


    # Remove comments from tokens

    comm_start = 0 # index at which comment starts
    in_comment = False
    for index, token in enumerate(tokens):
        if token == comment_start and not in_comment:
            in_comment = True
            comm_start = index
        if token == comment_end and in_comment:
            in_comment = False

             # remove tokens in the comment (replace with None)
            for i in range(comm_start, index+1): tokens[i] = None

    tokens = [token for token in tokens if token]

    # stores the stack of symbols for the bottom-up shift-reduce parser
    stack = []
    # stores the tree itself in an analogous stack
    tree_stack = []

    while True:
        #print(stack) # great for debugging
```

```
        skip_neg = False # don't apply the unary +/- rule if binary +/- skipped
        skip_point = False # don't apply pointer rule if binary * skipped

        for rule in rules:

            if len(rule.new) > len(stack): continue
            else:
                # check if the rule matches with the top of the stack
                for rule_el, stack_el in zip(reversed(rule.new), reversed(stack)):
                    # Break if any element of rule doesn't match the stack
                    if not rule_el.match(stack_el): break
                else:

                    if( rule.priority is not None
                        and len(tokens) > 0 and tokens[0].priority is not None
                        and tokens[0].priority > rule.priority ):

                        if rule == add_rule: skip_neg = True
                        # If we skiped binary * for this reason, also skip pointer dereference
                        if rule == mult_rule: skip_point = True

                    # if we're supposed to skip unary +/-, do so
                    elif rule == neg_rule and skip_neg:
                        pass
                    # if we're supposed to skip unary *, do so
                    elif rule == pointer_rule and skip_point:
                        pass

                    elif rule == dec_sep_rule and stack[-2] != dec_exp_symbol:
                        pass

                    else:
```

```
                        tree_stack = tree_stack[:-len(rule.new)] +
[ParseNode(rule, tree_stack[-len(rule.new):])]
                        # simplify the stack
                        stack = stack[:-len(rule.new)] + [rule.orig]
                        break # don't bother checking the rest of the rules
        else: # none of the rules matched
            # if we're all out of tokens, we're done
            if not tokens: break
            else: # inject another token into the stack
                stack.append(tokens[0])
                tree_stack.append(tokens[0])
                tokens = tokens[1:]

    # when we're done, we should have the start symbol left in the stack
    if stack == [start_symbol]:
        return tree_stack[0]
    else:
        raise ParseException(tree_stack)
```

/* Parser Code ends */

☑ Task 4:

Write a PARSER program to transform the list of tokens into the syntax tree.

/* Generating Assembly Code *(code_gen.py)* */

```
class RuleGenException(Exception):
   """If code generation fails for a particular rule"""
   def __init__(self, rule):
       self.rule = rule
   def __str__(self):
       return "Problem generating code for rule: " + str(self.rule)

class VariableRedeclarationException(Exception):
   """If a variable is defined multiple times"""
   def __init__(self, name):
       self.name = name
   def __str__(self):
       return "Variable declared multiple times: " + self.name

class VariableNotDeclaredException(Exception):
   """If a variable is not declared before it's used"""
   def __init__(self, name):
       self.name = name
   def __str__(self):
       return "Variable not declared: " + self.name

class CodeManager:

   def __init__(self):
       # Add all the starting boilerplate assembly
       self.setup = ["\tglobal start",
                     "",
                     "\tsection .text",
                     "",
                     "start:",
```

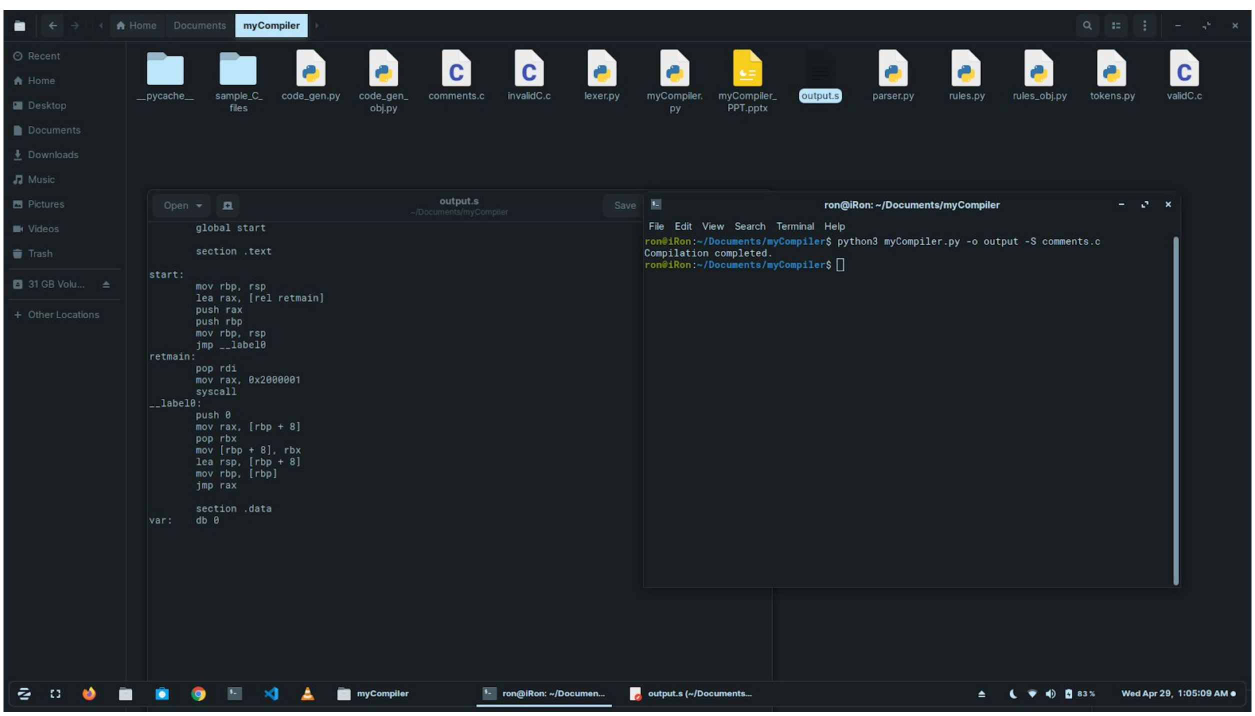
```
                    "\tmov rbp, rsp",
                    "\tlea rax, [rel retmain]",
                    "\tpush rax",
                    "\tpush rbp",
                    "\tmov rbp, rsp",
                    "", # on calling get_code, make this "\tjmp
main_label"
                    "retmain:",
                    "\tpop rdi",
                    "\tmov rax, 0x2000001",
                    "\tsyscall"]
        self.lines = [] # stores the asm lines generated
        self.data = ["", "\tsection .data", "var:\tdb 0"]

        self.labelnum = 0 # current label number
    def get_code(self, main_label):

        self.setup[10] = "\tjmp " + main_label
        return '\n'.join(self.setup + self.lines + self.data)
    def add_command(self, comm, arg1 = "", arg2 = ""):
        """Adds a command to the list of current commands"""
        self.lines.append("\t"+ comm +
                          ((" " + arg1) if arg1 else "") +
                          ((", " + arg2) if arg2 else ""))
    def get_label(self):
        """Returns a string that is an unused label name"""
        label_name = "__label" + str(self.labelnum)
        self.labelnum += 1
        return label_name

    def add_label(self, label_name):

        self.lines.append(label_name + ":")

class Type:

    def __init__(self, type_name = "int", pointers = 0):
```

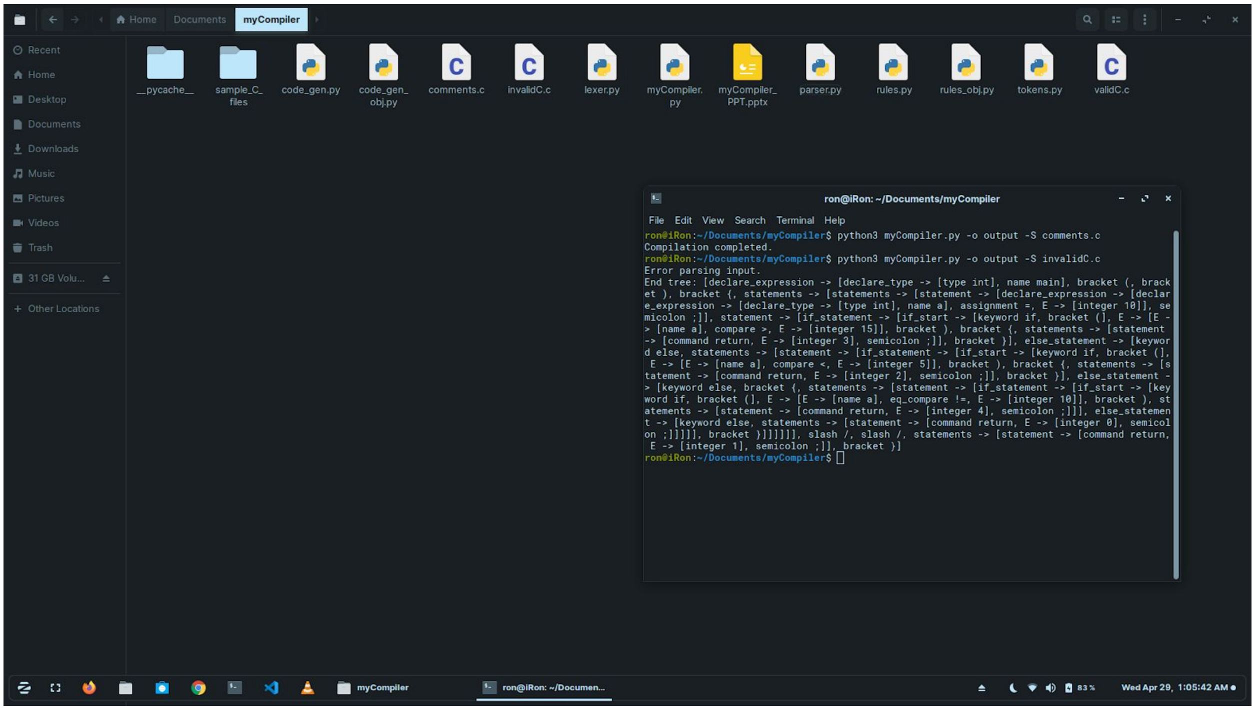
```
        self.type_name = type_name
        self.pointers = pointers
    def __repr__(self):
        return "*"*self.pointers + self.type_name
    def __eq__(self, other):
        return (self.type_name == other.type_name) and (self.pointers ==
other.pointers)

class StateInfo:

    def __init__(self, var_offset = 0, symbols = [], funcs = [], t =
Type()):
        # Amount of offset from rbp for last variable, divided by 8
        self.var_offset = var_offset
        self.symbols = symbols[:] # symbol table
        self.funcs = funcs[:] # function table

        # the type of the most thing returned by the most recent node
        # (see code_gen.py for more explanation)
        self.t = t
    def is_declared(self, name):
        """Whether a variable is declared"""
        return (name in [dec[0] for dec in self.symbols])
    def func_declared(self, func_name):
        """Whether a function is declared"""
        return (func_name in [func["fname"] for func in self.funcs])
    def add_space(self):

        s = self.c()
        s.var_offset += 1
        return s
    def add(self, name, t):

        if self.is_declared(name): raise
VariableRedeclarationException(name)
        else:
            s = self.c()
```

```
            s.var_offset += 1
            s.symbols += [(name, s.var_offset, t)]
            return s
    def get(self, name):
        """Returns the memory location and type of a variable"""
        var_loc = [var for var in self.symbols if var[0] == name]
        if var_loc:
            return (var_loc[0][1], var_loc[0][2])
        else:
            raise VariableNotDeclaredException(name)
    def add_func(self, func_name, func_type, func_args, label):
        """Returns a copy of StateInfo with a function added to the
function table"""
        if self.func_declared(func_name): raise
VariableRedeclarationException(func_name)
        else:
            s = self.c()
            s.funcs += [{"fname": func_name,
                         "ftype": func_type,
                         "args": func_args,
                         "label": label}] #TODO: complain if function is
declared but never defined
            return s
    def get_func(self, func_name):
        """Returns details about the function `func_name`"""
        func_loc = [func for func in self.funcs if func["fname"] ==
func_name]
        if func_loc: return func_loc[0]
        else:
            raise VariableNotDeclaredException(func_name)
    def c(self):
        """Returns a copy of this object"""
        return StateInfo(self.var_offset, self.symbols, self.funcs, self.t)
```

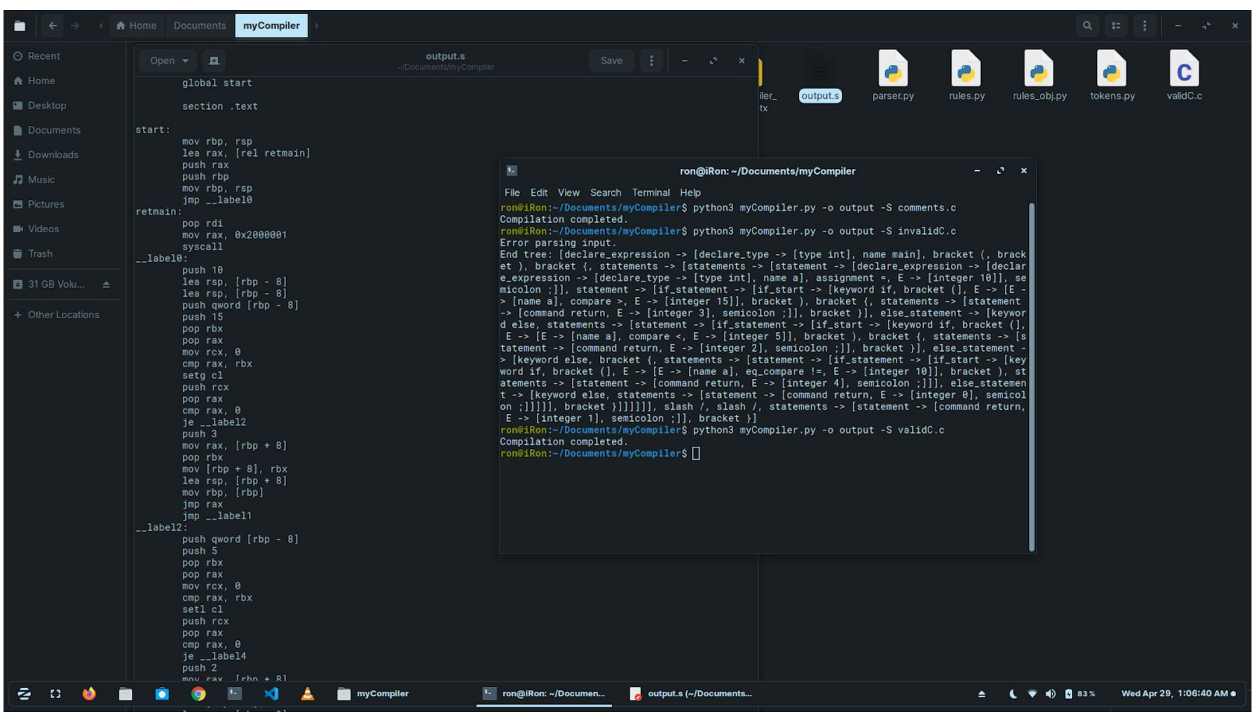
/* Generating Assembly Code ends */

/* myCompiler - Usage Screenshots */





Home
Documents
myCompiler
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
31 GB Volu...
Other Locations
__pycache__
sample_C_files
code_gen.py
code_gen_obj.py
comments.c
invalidC.c
lexer.py
myCompiler.py
myCompiler_PPT.pptx
parser.py
rules.py
rules_obj.py
tokens.py
validC.c
ron@iRon: ~/Documents/myCompiler
File Edit View Search Terminal Help
ron@iRon:~/Documents/myCompiler$ python3 myCompiler.py -o output -S comments.c
Compilation completed.
ron@iRon:~/Documents/myCompiler$ python3 myCompiler.py -o output -S invalidC.c
Error parsing input.
End tree: [declare_expression -> [declare_type -> [type int], name main], bracket (, bracket), bracket {, statements -> [statements -> [statement -> [declare_expression -> [declare_expression -> [declare_type -> [type int], name a], assignment =, E -> [integer 10]], semicolon ;]], statement -> [if_statement -> [if_start -> [keyword if, bracket (], E -> [E -> [name a], compare >, E -> [integer 15]], bracket), bracket {, statements -> [statement -> [command return, E -> [integer 3], semicolon ;]], bracket }], else_statement -> [keyword else, statements -> [statement -> [if_statement -> [if_start -> [keyword if, bracket (], E -> [E -> [name a], compare <, E -> [integer 5]], bracket), bracket {, statements -> [statement -> [command return, E -> [integer 2], semicolon ;]], bracket }], else_statement -> [keyword else, bracket {, statements -> [statement -> [if_statement -> [if_start -> [keyword if, bracket (], E -> [E -> [name a], eq_compare !=, E -> [integer 10]], bracket), statements -> [statement -> [command return, E -> [integer 4], semicolon ;]]], else_statement -> [keyword else, statements -> [statement -> [command return, E -> [integer 0], semicolon ;]]]]], bracket }]]]]]], slash /, slash /, statements -> [statement -> [command return, E -> [integer 1], semicolon ;]], bracket }]
ron@iRon:~/Documents/myCompiler$
myCompiler
ron@iRon: ~/Documen...
83 %
Wed Apr 29, 1:05:42 AM



Home
Documents
myCompiler
Open
output.s
~/Documents/myCompiler
Save
global start
section .text
start:
mov rbp, rsp
lea rax, [rel retmain]
push rax
push rbp
mov rbp, rsp
jmp __label0
retmain:
pop rdi
mov rax, 0x2000001
syscall
__label0:
push 10
lea rsp, [rbp - 8]
lea rsp, [rbp - 8]
push qword [rbp - 8]
push 15
pop rbx
pop rax
mov rcx, 0
cmp rax, rbx
setg cl
push rcx
pop rax
cmp rax, 0
je __label2
push 3
mov rax, [rbp + 8]
pop rbx
mov [rbp + 8], rbx
lea rsp, [rbp + 8]
mov rbp, [rbp]
jmp rax
jmp __label1
__label2:
push qword [rbp - 8]
push 5
pop rbx
pop rax
mov rcx, 0
cmp rax, rbx
setl cl
push rcx
pop rax
cmp rax, 0
je __label4
push 2
output.s
parser.py
rules.py
rules_obj.py
tokens.py
validC.c
ron@iRon: ~/Documents/myCompiler
File Edit View Search Terminal Help
ron@iRon:~/Documents/myCompiler$ python3 myCompiler.py -o output -S comments.c
Compilation completed.
ron@iRon:~/Documents/myCompiler$ python3 myCompiler.py -o output -S invalidC.c
Error parsing input.
ron@iRon:~/Documents/myCompiler$ python3 myCompiler.py -o output -S validC.c
Compilation completed.
ron@iRon:~/Documents/myCompiler$
myCompiler
ron@iRon: ~/Documen...
output.s (~/Documents...
83 %
Wed Apr 29, 1:06:40 AM