Processamento de Linguagens (3º ano de Curso)

Trabalho Prático 2

Relatório de Desenvolvimento

Paulo Silva Sousa (a89465)

João Figueiredo Martins Peixe dos Santos (a89520)

Luis Filipe Cruz Sobral (a89474)

30 de maio de 2021

Resumo

Neste trabalho iremos desenvolver um compilador gerando código para uma máquina de stack virtual, através do uso de um gerador de analisadores léxicos e gerador de compiladaores, mais concretamente através do Lex/Yacc do Ply/Python.

Índice

Capítulo 1

Introdução

Supervisor: Pedro Rangel Henriques

1.1 GIC/GT + Compiladores

Área: Processamento de Linguagens

No âmbito da unidade curricular de Processamento de Linguagens, foi nos proposto um desenvolvimento de um último projeto descrito no presente relatório.
Deste modo, o nosso grupo, para além de criar a sua própria linguagem, também desenvolveu um compilador para esta. Como objetivos para este projeto temos o desenvolvimento de processadores de linguagens sobre o método dirigido pela sintaxe e de um compilador que gera código para uma máquina de stack virtual, uma maior experiência de engenharia de linguagens e em programação generativa e, por último, o uso de geradores de compiladores baseados em gramáticas tradutores completado pelo gerador de analisadores léxicos.

1.2 Estrutura do Relatório

Este relatório está dividido em 4 partes:

- No segundo capítulo é apresentada uma análise e descrição do problema, começando por uma descrição informal e passando depois para os dados e pedidos específicos.
- O terceiro capítulo é dedicado ao desenho e concepção da resolução, onde apresentamos a nossa estratégia para desenvolver o problema
- No quarto capítulo apresentamos os testes e resultados do nosso problema
- No quinto capítulo fazemos uma reflexão crítica de todo o trabalho realizado

Além disso, ainda é apresentado o código gerado pelo compilador e o código do compilador em si.

Capítulo 2

Análise e especificação

2.1 Descrição informal do problema

Pretende-se que comece por definir uma linguagem de programação imperativa simples, a seu gosto.

Como é da praxe neste tipo de linguagens, as variáveis deverão ser declaradas no início do programa e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explicitado, o valor da variável após a declaração é 0 (zero).

Desenvolva, então, um compilador para essa linguagem com base na GIC criada acima e com recurso aos módulos Yacc/Lex do PLY/Python.

O compilador deve gerar pseudo-código, Assembly da Máquina Virtual VM.

2.2 Especificação dos Requisitos

2.2.1 Dados

Apenas deve ter em consideração que essa linguagem terá de permitir:

- declararvariáveis atómicas do tipo inteiro, com os quais se podem realizar as habituais operações aritméticas ,relacionais e lógicas.
- efetuar instruções algorítmicas básicas como a atribuição do valor de expressões numéricas a variáveis.
- ler do standard input e escrever no standard output.
- efetuar instruções condicionais para controlo do fluxo de execução.
- efetuar instruções cíclicas para controlo do fluxo de execução, permitindo o seu aninhamento. Note que deve implementar pelo menos o ciclo while-do, repeat-until ou for-do conforme o Número do seu Grupo módulo 3 seja 0, 1 ou 2.

Adicionalmente deve ainda suportar, à sua escolha, uma das duas funcionalidades seguintes:

- declarar e manusearvariáveis estruturadas do tipoarray (a 1 ou 2 dimensões) de inteiros, em relação aos quais é apenas permitida a operação de indexação (índice inteiro).
- definir e invocar subprogramassem parâmetros mas que possam retornar um resultado do tipo inteiro.

2.2.2 Pedidos

Para a entrega do TP deve preparar um conjunto de testes (programas-fonte escritos na sua linguagem) e mostrar o código Assembly gerado bem como o programa a correr na máquina virtual VM. Esse conjunto terá de conter, no mínimo, os 4 primeiros exemplos abaixo e um dos 2 últimos conforme a sua escolha acima:

- ler 4 números e dizer se podem ser os lados de um quadrado.
- ler um inteiro N, depois ler N números e escrever o menor deles.
- ler N (constante do programa) nu meros e calcular e imprimir o seu produtório.
- contar e imprimir os números impares de uma sequência de números naturais.
- ler e armazenar N números num array; imprimir os valores por ordem inversa.
- invocar e usar num programa seu uma função 'potencia()', que começa por ler do input a base B e o expoente E e retorna o valor B^E.

Capítulo 3

Desenho e concepção da resolução

3.1 Concepção da Linguagem

De modo a desenvolvermos a linguagem de programação imperativa para a realização do trabalho, decidimos utilizar uma syntax simples e fácil de ler.

Para isso, decidimos seguir a syntax da linguagem de programação Ruby. Nesta linguagem, todos os blocos de código, como por exemplo instruções condicionais, ciclicas ou funções são delimitados por um *end* em vez de chavetas. Esta opção torna o código mais limpo e a leitura mais simplificada.

Uma das maiores diferenças entre a nossa linguagem e Ruby é a existencia de ponto e virgula para delimitar cada linha de codigo, uma vez que achamos que este delimitar torna a linguagem mais fácil de ler.

3.2 Lex

3.2.1 Tokens e Palavras Reservadas

No nosso ficheiro lex definimos como tokens, além do ponto e virgula e dos parenteses retos e curvos, todos os operados relacionais e aritméticos.

Para além disso, definimos um *ID*, que pode ser utilizado como identificador de uma variável ou como nome de uma função, um *NUM*, que representa um número inteiro e uma *STRING*, para ser possível imprimir mensagens de texto na nossa aplicação.

```
tokens = ['ID', 'PLUS', 'MINUS',
          'MUL', 'DIV', 'MOD',
          'EQ', 'NE', 'G', 'GE',
          'L', 'LE', 'LPAR',
          'RPAR', 'LBRA', 'RBRA',
          'SC', 'ATTR', 'NUM',
          'STRING'] + list(reserved.values())
```

Além dos tokens, definimos também palavras reservadas, de modo a tornar o nosso compilador mais robusto contra utilização de palavras específicas da linguagem como nomes de variáveis, por exemplo.

```
reserved = {
    'int' : 'INT',
    'print' : 'PRINT',
    'input' : 'INPUT',
    'then' : 'THEN',
    'end': 'END',
    'declarations': 'DECLARATIONS',
    'instructions': 'INSTRUCTIONS',
    'if' : 'IF',
    'else' : 'ELSE',
    'repeat' : 'REPEAT',
    'until' : 'UNTIL',
    'while' : 'WHILE',
    'call' : 'CALL',
    'for' : 'FOR',
    'do' : 'DO',
    'and' : 'AND',
    'or' : 'OR',
    'not' : 'NOT',
    'def' : 'DEF',
    'main' : 'MAIN'
}
```

3.3 Yacc

3.3.1 Main e Funções

Primeiramente, decidimos dividir o nosso programa em duas secções, a Main e as restantes funções.

```
def p_Program(p):
    "Program : Main Functions"
    p[0] = p[1] + p[2]
```

A Main tem obrigatóriamente um bloco de declarações, onde são declaradas todas as variáveis que possam ser utilizadas no programa, e um bloco de intruções, onde são executadas todas as instruções, como por exemplo, atribuições, escrita e leitura de variávies, chamadas de funções, etc. Deste modo, a Main apenas escreve o comando *start* após a declaração das variáveis e escreve o comando *stop* após todas as instruções desta serem executadas.

```
def p_Main(p):
    "Main : DEF MAIN MainDeclarations MainInstructions END"

    p[0] = f"{p[3]}start\n{p[4]}stop\n"

def p_Main_declarations(p):
    "MainDeclarations : DECLARATIONS Declarations END"

    p[0] = p[2]

def p_Main_instructions(p):
    "MainInstructions : INSTRUCTIONS Instructions END"

    p[0] = p[2]

def p_Declarations(p):
    "Declarations : Declarations Declaration"

    p[0] = p[1] + p[2]

def p_Declarations_empty(p):
    "Declarations : "

    p[0] = ""

def p_Instructions(p):
    "Instructions : Instructions Instruction"

    p[0] = p[1] + p[2]

def p_Instructions_empty(p):
    "Instructions : "
```

```
    p[0] = ""
```

As funções são obrigatóriamente definidas depois da Main. A inclusão destas num programa é opcional e ausenta qualquer tipo de declaração de variáveis, utilizando as variávies definidas na Main. Em cada função são executadas todas as intruções seguido do comando *return*, que nos leva para o sitio onde a função foi chamada.
Quando uma função é reconhecida, o seu nome é adicionado a um set (*parser.funcs*) para, mais tarde, haver controlo de erros.

```
def p_Functions(p):
    "Functions : Functions Function"
    p[0] = p[1] + p[2]

def p_Functions_empty(p):
    "Functions : "
    p[0] = ""

def p_Function(p):
    "Function : DEF ID Instructions END"

    add_func(p[2], p)

    p[0] = f"{p[2]}:\n{p[3]}return\n"
```

Para ser possível uma função ser chamada na nossa aplicação criamos uma instrução que, utilizando a palavra reservada call, chama a função desejada. O nome da função que é chamada é adicionado a um set (*parser.called_funcs*) para posterior controlo de erros. Este processo é realizado fazendo *pusha* do nome da variável, seguido da instrução *call*.

```
def p_Call(p):
    "Instruction : CALL ID SC"

    add_func_called(p[2], p)

    p[0] = f"pusha {p[2]}\ncall\n"
```

3.3.2 Declaração de Variávies e Arrays

Na nossa linguagem é possível existir múltiplas declarações, podendo ser estas de variáveis, arrays ou matrizes.

A declaração de variáveis pode ser realizada de duas maneiras: através da atribuição de uma expressão ou ausentando a atriuição, sendo a variável declarada com o valor 0.

Quando uma variável é declarada é guardado num dicionário de registos (*parser.var*) o offset da variável na stack, para esta poder ser posteriormente acedida. Além disso, é retornado o comando *pushi* seguido do valor atriuido.

```
def p_Declararation_exp(p):
    "Declaration : INT ID ATTR Exp SC"

    add_var(p[2], 1, 1, p)

    p[0] = p[4]

def p_Declararation_simple(p):
    "Declaration : INT ID SC"

    add_var(p[2], 1, 1, p)

    p[0] = "pushi 0\n"
```

Para declarar arrays e matrizes o processo é similar. É guardado no mesmo dicionário o offset, adicionando também o número de linhas e colunas (arrays são inicializados com colunas a 1). Como é utilizado o comando *pushn* seguido do tamanho do array/matriz, não é possível atribuir nenhum valor na inicialização, ficando todas as posições com o valor 0.

```
def p_Declararation_array_num(p):
    "Declaration : INT ID LBRA NUM RBRA SC"

    col = int(p[4])
    add_var(p[2], col, 1, p)

    p[0] = f"pushn {col}\n"


def p_Declararation_matrix_num(p):
    "Declaration : INT ID LBRA NUM RBRA LBRA NUM RBRA SC"

    add_var(p[2], int(p[7]), int(p[4]), p)

    size = int(p[7]) * int(p[4])
    p[0] = f"pushn {size}\n"
```

3.3.3 Operações aritméticas, relacionais e lógicas

De modo a realizar operações aritméticas, relacionais e lógicas na nossa linguagem, utilizamos a seguinte precedências de operações:

- **Parênteses** ( )
- **Multiplicativos** * / %
- **Aditivos** + -
- **Igualitários** == !=
- **Relacionais** $><\leq\geq$
- **Logicos** and or not

Sendo assim, adaptamos o código utilizado nas aulas (Exp, Termo e Fator) e adicionamos os novos operadores.

```
def p_Log_and(p):
    "Log : Log AND Rel"
    p[0] = f"{p[1]}{p[3]}mul\n"

def p_Log_or(p):
    "Log : Log OR Rel"
    p[0] = f"{p[1]}{p[3]}add\n{p[1]}{p[3]}mul\nsub\n"

def p_Log_not(p):
    "Log : NOT Rel"
    p[0] = "{p[2]}not\n"

def p_Log_Rel(p):
    "Log : Rel"
    p[0] = p[1]

def p_Rel_g(p):
    "Rel : Rel G Rel2"
    p[0] = f"{p[1]}{p[3]}sup\n"

def p_Rel_ge(p):
    "Rel : Rel GE Rel2"
    p[0] = f"{p[1]}{p[3]}supeq\n"

def p_Rel_l(p):
    "Rel : Rel L Rel2"
    p[0] = f"{p[1]}{p[3]}inf\n"

def p_Rel_le(p):
    "Rel : Rel LE Rel2"
```

```
    p[0] = f"{p[1]}{p[3]}infeq\n"

def p_Rel_rel2(p):
    "Rel : Rel2"
    p[0] = p[1]

def p_Rel2_eq(p):
    "Rel2 : Rel2 EQ Exp"
    p[0] = f"{p[1]}{p[3]}equal\n"

def p_Rel2_ne(p):
    "Rel2 : Rel2 NE Exp"
    p[0] = f"{p[1]}{p[3]}equal\nnot\n"

def p_Rel2_lexp(p):
    "Rel2 : Exp"
    p[0] = p[1]

def p_Exp_add(p):
    "Exp : Exp PLUS Term"
    p[0] = f"{p[1]}{p[3]}add\n"

def p_Exp_sub(p):
    "Exp : Exp MINUS Term"
    p[0] = f"{p[1]}{p[3]}sub\n"

def p_Exp_term(p):
    "Exp : Term"
    p[0] = p[1]

def p_Term_mult(p):
    "Term : Term MUL Factor"
    p[0] = f"{p[1]}{p[3]}mul\n"

def p_Term_div(p):
    "Term : Term DIV Factor"
    p[0] = f"{p[1]}{p[3]}div\n"

def p_Term_mod(p):
    "Term : Term MOD Factor"
    p[0] = f"{p[1]}{p[3]}mod\n"

def p_Term_factor(p):
    "Term : Factor"
    p[0] = p[1]

def p_Factor_par(p):
    "Factor : LPAR Log RPAR"
```

```
    p[0] = p[2]

def p_Factor_num(p):
    "Factor : NUM"
    p[0] = f"pushi {p[1]}\n"

def p_Factor_id(p):
    "Factor : ID"
    res = get_index(p[1], p)

    (_, _, offset) = res
    p[0] = f"pushg {offset}\n"

def p_Factor_array(p):
    "Factor : ID LBRA Exp RBRA"

    res = get_index(p[1], p)

    (_, _, offset) = res
    p[0] = f"pushgp\npushi {offset}\npadd\n{p[3]}loadn\n"

def p_Factor_matrix(p):
    "Factor : ID LBRA Exp RBRA LBRA Exp RBRA"

    res = get_index(p[1], p)

    (col, _, offset) = res
    p[0] = f"pushgp\npushi {offset}\npadd\n{p[3]}pushi {col}\nmul\n{p[6]}add\nloadn\n"
```

Quando estamos reduzidos a um fator estamos reduzidos a 5 opções.

Caso seja uma expressão dentro de parênteses voltamos a reduzir a Log.

Caso contrário, temos de retornar o comando push. No caso de se tratar de um número, fazemos *pushi* do seu valor. Já no caso de uma variável temos de fazer *pushg* do seu offset na stack. Por ultimo, no caso de arrays e matrizes, temos de utilizar os comandos *pushgp*, *padd*, *loadn*, entre outros, para aceder à posição de memória correspondente ao seu indice.

3.3.4 Atribuição de valores a variávies e arrays

Para atribuir valores a variáveis e arrays limitamonos a guardar o resultado de uma expressão na sua posição de memória.
No caso de uma variável, apenas utilizamos o comando *storeg* para guardar o resultado proveniente da expressão no offset da variável.

```
def p_Attribure(p):
    "Instruction : ID ATTR Exp SC"

    res = get_index(p[1], p)

    (_, _, offset) = res
    p[0] = f"{p[3]}storeg {offset}\n"
```

No caso de arrays e matrizes, tal como nas expressões, temos de recorrer aos comandos *pushgp* e *padd* para calcularmos o endereço correto, fazendo no fim *storen*.

```
def p_Attribure_array(p):
    "Instruction : ID LBRA Exp RBRA ATTR Exp SC"

    res = get_index(p[1], p)

    (_, _, offset) = res

    p[0] = f"pushgp\npushi {offset}\npadd\n{p[3]}{p[6]}storen\n"

def p_Attribure_matrix(p):
    "Instruction : ID LBRA Exp RBRA LBRA Exp RBRA ATTR Exp SC"

    res = get_index(p[1], p)

    (col, _, offset) = res

    p[0] = f"pushgp\npushi {offset}\npadd\n{p[3]}pushi {col}\nmul\n{p[6]}add\n{p[9]}storen\n"
```

3.3.5 Leitura do Stdin e Escrita no Stdout

Quando se trata de ler do Stdin, necessitamos para todos os casos de realizar um *read* que recebe o input do utilizador seguido de um *atoi* que converte a String num inteiro. O processo de armazenamento numa variável ou array é similar ao do tópico anteiror, atribuição de valores.

```
def p_Read_id(p):
    "Instruction : INPUT ID SC"

    res = get_index(p[2], p)

    (_, _, offset) = res
    p[0] = f'''pushs ">> "\nwrites\nread\natoi\nstoreg {offset}\n'''

def p_Read_array(p):
    "Instruction : INPUT ID LBRA Exp RBRA SC"

    res = get_index(p[2], p)

    (_, _, offset) = res
    p[0] = f'''pushs ">> "\nwrites\npushgp\npushi {offset}\npadd\n{p[4]}
    read\natoi\nstoren\n'''

def p_Read_matrix(p):
    "Instruction : INPUT ID LBRA Exp RBRA LBRA Exp RBRA SC"

    res = get_index(p[2], p)

    (col, _, offset) = res
    p[0] = f'''pushs ">> "\nwrites\npushgp\npushi {offset}\npadd\n{p[4]}pushi
    {col}\nmul\n{p[7]}add\nread\natoi\nstoren\n'''
```

Em relação à escrita no Stdout temos duas opções: escrita de uma expressão, que engloba, para além de expressões, escrita de números, variáveis e arrays; e escrita de uma string. No caso da escrita de expressões é utilizado o comando *writei*, enquanto que na escrita de expressões utilizamos os comandos *pushs* e *writes*.

```
def p_Print(p):
    "Instruction : PRINT Line SC"
    p[0] = p[2]

def p_Line_String(p):
    "Line : STRING"

    p[0] = f'''pushs {p[1]}\nwrites\npushs "\\n"\nwrites\n'''

def p_Line_Log(p):
    "Line : Log"

    p[0] = f'''{p[1]}writei\npushs "\\n"\nwrites\n'''
```

3.3.6 Instruções Condicionais

Para escrita de expressões condicionais na nossa linguagem, consideramos duas hipóteses. Utilização de apenas um if e utilização de um if e de um else. Para podermos realizar os saltos necessários, guardamos um contador do numero de ifs no programa, ficando cada instrução condicional com um "id"associado.

No primeiro caso, no momento da entrada na instrução é testada uma condição. Se esta for falsa, é executado o comando *jz* para saltar para o fim do if. Caso contrário, são executadas as instruções dentro do bloco de código.

```
def p_Condition_simple(p):
    "Instruction : IF LPAR Log RPAR THEN Instructions END"
    p[0] = f"{p[3]}jz endif{p.parser.ifs}\n{p[6]}\nendif{p.parser.ifs}:\n"
    p.parser.ifs += 1
```

No caso de existir um else, os saltos funcionam de maneira diferente. Após a condição, é realizado um *jz* para o inicio do código do else, caso esta seja falsa, e são executadas as instruções do if caso contrário.

No fim do código do if, é utilizado o comando *jump* para o fim de toda a instrução condicinal de modo a não ser executado o código do else.

```
def p_Condition(p):
    "Instruction : IF LPAR Log RPAR THEN Instructions ELSE Instructions END"
    p[0] = f"{p[3]}jz else{p.parser.ifs}\n{p[6]}jump
    endif{p.parser.ifs}\nelse{p.parser.ifs}:\n{p[8]}\nendif{p.parser.ifs}:\n"
    p.parser.ifs += 1
```

3.3.7 Instruções Cíclicas

Segundo o enunciado, para as instruções ciclicas, foi nos atribuido o ciclo 1 (61 % 3), que corresponde ao repeat-until. No entanto, o grupo decidiu desenvolver os três ciclos.

Reapeat Until

Este ciclo tem a propriedade da condição ser testada no fim do ciclo. Para isso, após cada iteração é testada a condição e caso esta seja falsa, é realizado um *jz* para o início do ciclo.

```
def p_Cycle_Repeat_Until(p):
    "Instruction : REPEAT Instructions UNTIL LPAR Log RPAR END"
    p[0] = f"cycle{p.parser.cycles}:\n{p[2]}{p[5]}jz cycle{p.parser.cycles}\n"
    p.parser.cycles += 1
```

While Do

Ao contrário do ciclo anterior, no ciclo while a condição é testada no início. Caso esta seja verdadeira entra no ciclo, caso contrário é realizado um *jz* para fora deste. Além disso, ainda é realizado um *jump* no fim de cada iteração para o início para a condição voltar a ser testada.

```
def p_Cycle_While_Do(p):
    "Instruction : WHILE LPAR Log RPAR DO Instructions END"
    p[0] = f"cycle{p.parser.cycles}:\n{p[3]}jz endcycle{p.parser.cycles}\n{p[6]}jump
    cycle{p.parser.cycles}\nendcycle{p.parser.cycles}:\n"
    p.parser.cycles += 1
```

For Do

Este ciclo segue os mesmos princípios que o ciclo while, mudando apenas o facto de existir uma atribuição antes do incio do ciclo e outra no fim de cada iteração.

```
def p_Cycle_For_Do(p):
    "Instruction : FOR LPAR ID ATTR Exp SC Log SC ID ATTR Exp RPAR DO Instructions END"

    res1 = get_index(p[3], p)
    (_, _, offset1) = res1

    res2 = get_index(p[9], p)
    (_, _, offset2) = res2

    p[0] = f"{p[5]}storeg {offset1}\ncycle{p.parser.cycles}:\n{p[7]}jz
    endcycle{p.parser.cycles}\n{p[14]}{p[11]}storeg {offset2}\njump
    cycle{p.parser.cycles}\nendcycle{p.parser.cycles}:\n"
    p.parser.cycles += 1
```

3.3.8 Controlo de erros

De modo a controlar os erros do nosso compilador tivemos em atenção alguns pormenores.
Primeiramente, de modo a não haver chamadas a funções inexistentes, criamos um método que verifica se o set *parser.called_funcs* é um subset do set *parser.funcs*. Caso seja falso, é levantada uma exceção.

```
def check_called_funcs(parser):
    for called in parser.called_funcs:
        if called not in parser.funcs:
            raise Exception
```

Além disso, cada vez que adicionamos o nome de uma função ao set de funções é verificado se esta já foi definida. Em caso afirmativo é levantada uma exceção.

```
def add_func(name, p):
    if name not in p.parser.funcs:
        p.parser.funcs.add(name)
    else:
        raise Exception
```

Do mesmo modo, quando adicionamos uma variável aos registos, é verificado se esta já existe. Caso exista, é também levantada uma exceção.

```
def add_var(id, col, line, p):
    if id not in p.parser.var.keys():
        p.parser.var[id] = (col, line, p.parser.offset)
        p.parser.offset += col * line
    else:
        raise Exception
```

Todas estas exceções são depois apanhadas juntamente com alguns erros de syntax quando se está a processar a escrita do ficheiro *.vm*. Em caso de erro, o ficheiro de output é removido.

```
try:
    result = parser.parse(data)
    fileOut.write(result)
    check_called_funcs(parser)
except (TypeError, Exception):
    print("\nCompilation error, aborting!")
    os.remove(outFilePath)
```

Capítulo 4

Testes

Neste capítulo iremos apresentar os testes realizados no desenvolvimento do projeto. O código para a máquina virtual gerado na compilação dos testes da linguagem encontra-se em anexo.

4.1 Teste 1

Ler 4 números e dizer se podem ser os lados de um quadrado.

```
def main
    declarations
        int i;
        int tamanho = 4;
        int lados[4];
    end
    instructions
        print "Introduza 4 lados";

        repeat
            input lados[i];
            i = i + 1;
        until (i==tamanho) end

        if(lados[0]==lados[1] and lados[0]==lados[2] and lados[0]==lados[3]) then
            print "Os 4 lados formam um quadrado";
        else
            print "Os 4 lados nao formam um quadrado";
        end
    end
end
```

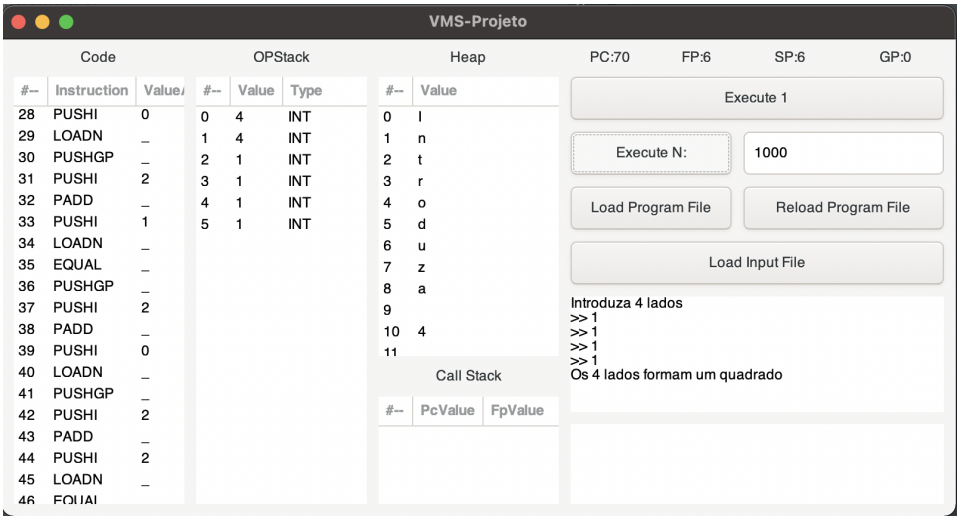

Figura 4.1: VMS teste 1

4.2 Teste 2

Ler um inteiro N, depois ler N números e escrever o menor deles.

```
def main
    declarations
        int n;
        int num;
        int min;
    end
    instructions
        print "Introduza um inteiro N:";

        input n;

        if(n>0) then
            print "Introduza N numeros:";

            input min;
            n = n - 1;

            if(n>0) then
                repeat
                    input num;
                    if(num<min) then
                        min = num;
                    end
                    n = n - 1;
                until (n==0) end
            end
```

```
            print "Numero menor:";
            print min;
        end
    end
end
```

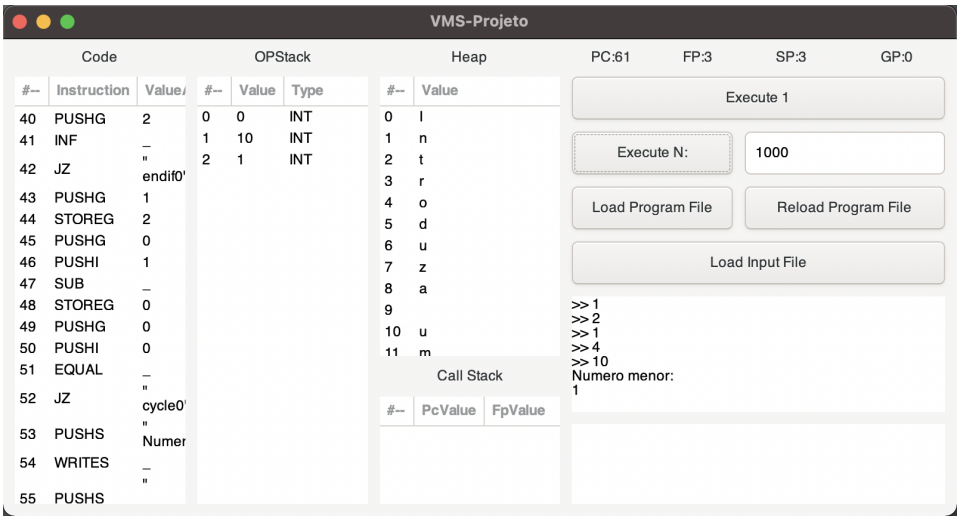

Figura 4.2: VMS teste 2

4.3 Teste 3

Ler N (constante do programa) nu meros e calcular e imprimir o seu produtório.

```
def main
    declarations
        int n=10;
        int prod=1;
        int aux;
    end
    instructions
        repeat
            input aux;
            prod = prod * aux;
            n = n-1;
        until(n==0) end

        print "Produtorio:";
        print prod;
    end
end
```

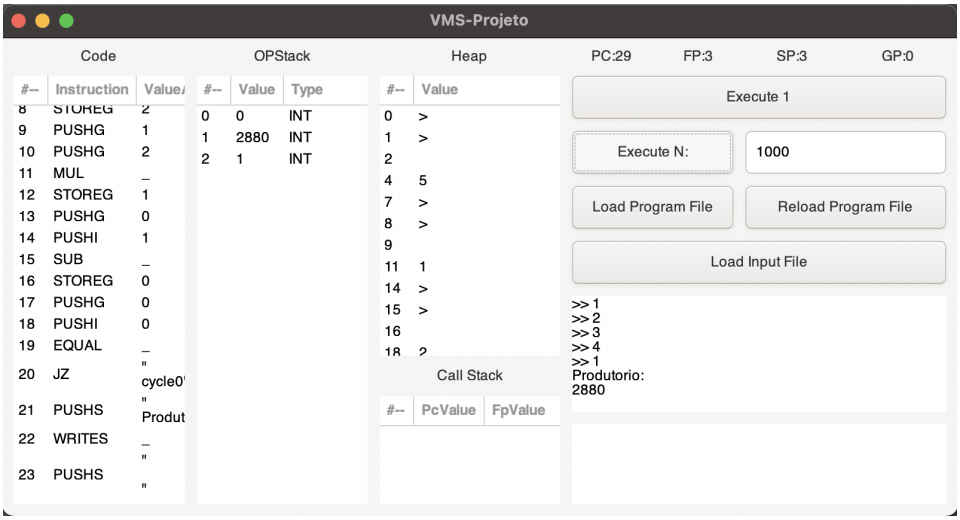

Figura 4.3: VMS teste 3

4.4 Teste 4

Contar e imprimir os números impares de uma sequência de números naturais.

```
def main
    declarations
        int n;
        int aux;
        int counter;
    end
    instructions
        print "Introduza um inteiro N:";
        input n;
        if(n>0) then
            print "Introduza numeros pares e impares";
            repeat
                input aux;
                if(aux % 2) then
                    print "Numero impar:";
                    print aux;
                    counter = counter + 1;
                end
                n = n-1;
            until(n==0) end

            print "Contador de numeros impares:";
            print counter;
        end
    end
end
```

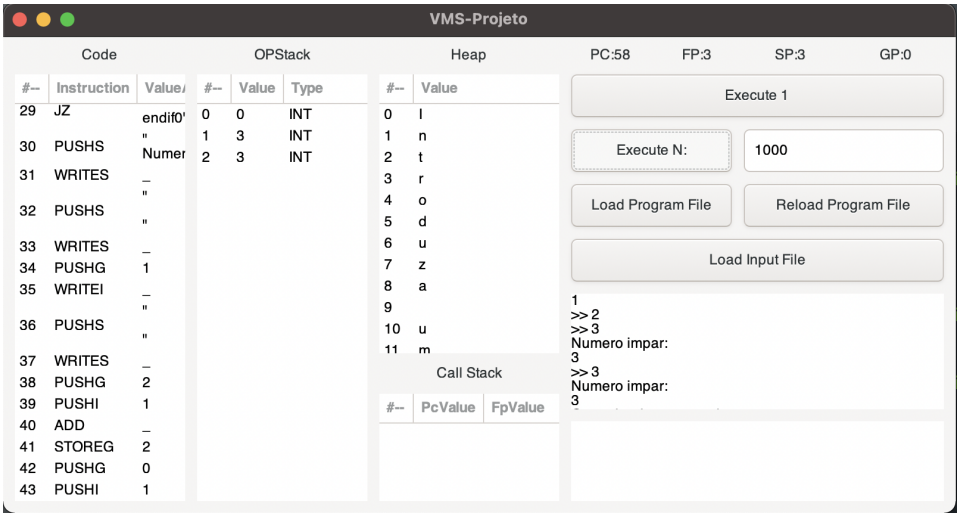

Figura 4.4: VMS teste 4

4.5 Teste 5

Ler e armazenar N números num array; imprimir os valores por ordem inversa.

```
def main
    declarations
        int n=10;
        int i;
        int arr[10];
    end
    instructions
        print "Introduza uma Sequencia de 10 numeros";
        repeat
            print "Introduza um numero";
            input arr[i];
            i = i + 1;
        until(i==n) end

        print "Sequencia invertida";
        repeat
            i = i - 1;
            print arr[i];
        until(i==0) end
    end
end
```

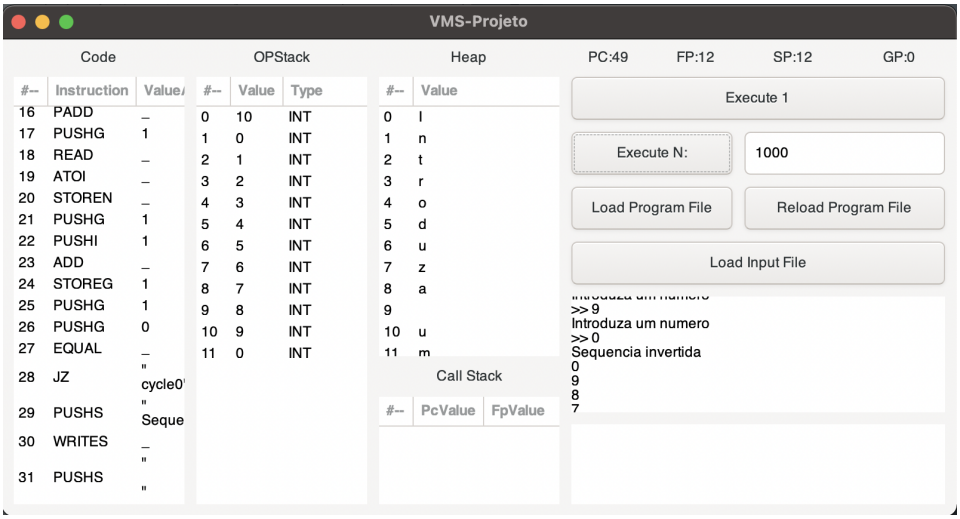

Figura 4.5: VMS teste 5

4.6 Teste 6

Invocar e usar num programa seu uma função 'potencia()', que começa por ler do input a base B e o expoente E e retorna o valor B^E.

```
def main
    declarations
        int res;
        int b;
        int e;
        int i=1;
    end
    instructions
        call potencia;

        print "Potencia: ";
        print res;
    end
end
```

```
def potencia
    print "Introduza uma base:";
    input b;
    print "Introduza um expoente:";
    input e;

    res = b;
    repeat
        res = res * b;
        i = i+1;
    until(i==e) end
end
```

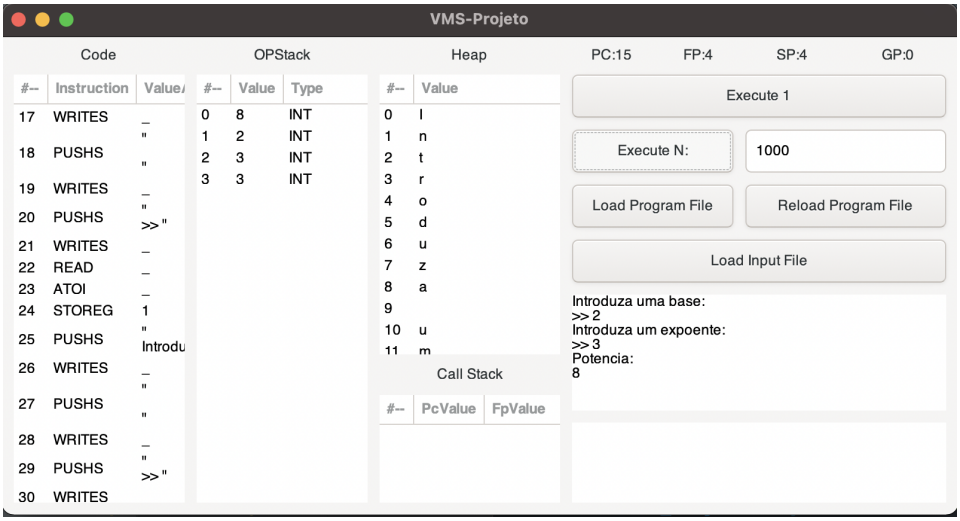

Figura 4.6: VMS teste 6

Capítulo 5

Conclusão

Para começar, consideramos que este último trabalho prático à Unidade Curricular de Processamento de Linguagens nos trouxe uma melhor consolidação da matéria dada nas aulas teóricas e práticas e um maior aproveitamento na UC.
Ao longo deste último projeto, conseguimos superar algumas dificuldades que se verificaram ao longo da realização deste, o que nos fez ganhar uma maior consolidação de conhecimentos e também um maior à vontade à UC de Processamento dde Linguagens.
Concluindo, o resumo que o grupo retira deste projeto é que foi bem-sucedido e cumpre todos os objetivos propostos, tanto por nós enquanto grupo como pelo que o enunciado impunha. Para além disso, adquirímos ais experiência no tema em questão, que se poderá tornar útil no futuro.

Apêndice A

Código Gerado

A.1 Teste 1

```
pushi 0
pushi 4
pushn 4
start
pushs "Introduza 4 lados"
writes
pushs "\n"
writes
cycle0:
pushs ">> "
writes
pushgp
pushi 2
padd
pushg 0
read
atoi
storen
pushg 0
pushi 1
add
storeg 0
pushg 0
pushg 1
equal
jz cycle0
pushgp
pushi 2
padd
pushi 0
loadn
pushgp
```

```
pushi 2
padd
pushi 1
loadn
equal
pushgp
pushi 2
padd
pushi 0
loadn
pushgp
pushi 2
padd
pushi 2
loadn
equal
mul
pushgp
pushi 2
padd
pushi 0
loadn
pushgp
pushi 2
padd
pushi 3
loadn
equal
mul
jz else0
pushs "Os 4 lados formam um quadrado"
writes
pushs "\n"
writes
jump endif0
else0:
pushs "Os 4 lados nao formam um quadrado"
writes
pushs "\n"
writes
endif0:
stop
```

A.2 Teste 2

```
pushi 0
pushi 0
pushi 0
start
pushs "Introduza um inteiro N:"
writes
pushs "\n"
writes
pushs ">> "
writes
read
atoi
storeg 0
pushg 0
pushi 0
sup
jz endif2
pushs "Introduza N numeros:"
writes
pushs "\n"
writes
pushs ">> "
writes
read
atoi
storeg 2
pushg 0
pushi 1
sub
storeg 0
pushg 0
pushi 0
sup
jz endif1
cycle0:
pushs ">> "
writes
read
atoi
storeg 1
pushg 1
pushg 2
inf
jz endif0
pushg 1
storeg 2
```

```
endif0:
pushg 0
pushi 1
sub
storeg 0
pushg 0
pushi 0
equal
jz cycle0
endif1:
pushs "Numero menor:"
writes
pushs "\n"
writes
pushg 2
writei
pushs "\n"
writes
endif2:
stop
```

A.3 Teste 3

```
pushi 10
pushi 1
pushi 0
start
cycle0:
pushs ">> "
writes
read
atoi
storeg 2
pushg 1
pushg 2
mul
storeg 1
pushg 0
pushi 1
sub
storeg 0
pushg 0
pushi 0
equal
jz cycle0
pushs "Produtorio:"
writes
pushs "\n"
writes
pushg 1
writei
pushs "\n"
writes
stop
```

A.4 Teste 4

```
pushi 0
pushi 0
pushi 0
start
pushs "Introduza um inteiro N:"
writes
pushs "\n"
writes
pushs ">> "
writes
read
atoi
storeg 0
pushg 0
pushi 0
sup
jz endif1
pushs "Introduza numeros pares e impares"
writes
pushs "\n"
writes
cycle0:
pushs ">> "
writes
read
atoi
storeg 1
pushg 1
pushi 2
mod
jz endif0
pushs "Numero impar:"
writes
pushs "\n"
writes
pushg 1
writei
pushs "\n"
writes
pushg 2
pushi 1
add
storeg 2
endif0:
pushg 0
pushi 1
```

```
sub
storeg 0
pushg 0
pushi 0
equal
jz cycle0
pushs "Contador de numeros impares:"
writes
pushs "\n"
writes
pushg 2
writei
pushs "\n"
writes
endif1:
stop
```

A.5 Teste 5

```
pushi 10
pushi 0
pushn 10
start
pushs "Introduza uma Sequencia de 10 numeros"
writes
pushs "\n"
writes
cycle0:
pushs "Introduza um numero"
writes
pushs "\n"
writes
pushs ">> "
writes
pushgp
pushi 2
padd
pushg 1
read
atoi
storen
pushg 1
pushi 1
add
storeg 1
pushg 1
pushg 0
equal
jz cycle0
pushs "Sequencia invertida"
writes
pushs "\n"
writes
cycle1:
pushg 1
pushi 1
sub
storeg 1
pushgp
pushi 2
padd
pushg 1
loadn
writei
pushs "\n"
```

```
writes
pushg 1
pushi 0
equal
jz cycle1
stop
```

A.6 Teste 6

```
pushi 0
pushi 0
pushi 0
pushi 1
start
pusha potencia
call
pushs "Potencia: "
writes
pushs "\n"
writes
pushg 0
writei
pushs "\n"
writes
stop
potencia:
pushs "Introduza uma base:"
writes
pushs "\n"
writes
pushs ">> "
writes
read
atoi
storeg 1
pushs "Introduza um expoente:"
writes
pushs "\n"
writes
pushs ">> "
writes
read
atoi
storeg 2
pushg 1
storeg 0
cycle0:
pushg 0
pushg 1
mul
storeg 0
pushg 3
pushi 1
add
storeg 3
```

```
pushg 3
pushg 2
equal
jz cycle0
return
```

Apêndice B

Código do Compilador

B.1 Lex

```
import ply.lex as lex

reserved = {
    'int' : 'INT',
    'print' : 'PRINT',
    'input' : 'INPUT',
    'then' : 'THEN',
    'end': 'END',
    'declarations': 'DECLARATIONS',
    'instructions': 'INSTRUCTIONS',
    'if' : 'IF',
    'else' : 'ELSE',
    'repeat' : 'REPEAT',
    'until' : 'UNTIL',
    'while' : 'WHILE',
    'call' : 'CALL',
    'for' : 'FOR',
    'do' : 'DO',
    'and' : 'AND',
    'or' : 'OR',
    'not' : 'NOT',
    'def' : 'DEF',
    'main' : 'MAIN'
}

tokens = ['ID', 'PLUS', 'MINUS', 'MUL', 'DIV', 'MOD', 'EQ', 'NE', 'G', 'GE', 'L', 'LE', 'LPAR',

def t_LPAR(t):
    r'\('
    t.type = reserved.get(t.value, 'LPAR')
    return t
```

```
def t_RPAR(t):
    r'\)'
    t.type = reserved.get(t.value, 'RPAR')
    return t

def t_LBRA(t):
    r'\['
    t.type = reserved.get(t.value, 'LBRA')
    return t

def t_RBRA(t):
    r'\]'
    t.type = reserved.get(t.value, 'RBRA')
    return t

def t_SC(t):
    r';'
    t.type = reserved.get(t.value, 'SC')
    return t

def t_C(t):
    r','
    t.type = reserved.get(t.value, 'C')
    return t

def t_ID(t):
    r'[a-z_][a-zA-Z0-9_]*'
    t.type = reserved.get(t.value,'ID')
    return t

def t_GE(t):
    r'>='
    t.type = reserved.get(t.value, 'GE')
    return t

def t_G(t):
    r'>'
    t.type = reserved.get(t.value, 'G')
    return t

def t_LE(t):
    r'<='
    t.type = reserved.get(t.value, 'LE')
    return t

def t_L(t):
    r'<'
    t.type = reserved.get(t.value, 'L')
```

```
    return t

def t_EQ(t):
    r'=='
    t.type = reserved.get(t.value, 'EQ')
    return t

def t_NE(t):
    r'!='
    t.type = reserved.get(t.value, 'NE')
    return t

def t_PLUS(t):
    r'\+'
    t.type = reserved.get(t.value, 'PLUS')
    return t

def t_MINUS(t):
    r'\-'
    t.type = reserved.get(t.value, 'MINUS')
    return t

def t_MUL(t):
    r'\*'
    t.type = reserved.get(t.value, 'MUL')
    return t

def t_DIV(t):
    r'/'
    t.type = reserved.get(t.value, 'DIV')
    return t

def t_MOD(t):
    r'\%'
    t.type = reserved.get(t.value, 'MOD')
    return t

def t_ATTR(t):
    r'='
    t.type = reserved.get(t.value, 'ATTR')
    return t

def t_NUM(t):
    r'\-?\d+'
    t.type = reserved.get(t.value, 'NUM')
    return t

def t_STRING(t):
```

```
    r'\"[^"]+\"'
    t.type = reserved.get(t.value, 'STRING')
    return t

t_ignore = " \r\t\n"

def t_error(t):
    print("Illegal character: ", t.value[0])
    t.lexer().skip(1)

lexer = lex.lex()
```

B.2 Yacc

```
# comp_yacc.py

import ply.yacc as yacc
from comp_lex import tokens
import os

# ------------------------------------------------------------------
#                            Program
# ------------------------------------------------------------------

def p_Program(p):
    "Program : Main Functions"
    p[0] = p[1] + p[2]

# ------------------------------------------------------------------
#                            MAIN
# ------------------------------------------------------------------

def p_Main(p):
    "Main : DEF MAIN MainDeclarations MainInstructions END"

    p[0] = f"{p[3]}start\n{p[4]}stop\n"

def p_Main_declarations(p):
    "MainDeclarations : DECLARATIONS Declarations END"

    p[0] = p[2]

def p_Main_instructions(p):
    "MainInstructions : INSTRUCTIONS Instructions END"

    p[0] = p[2]

# ------------------------------------------------------------------
#                          Functions
# ------------------------------------------------------------------

def p_Functions(p):
    "Functions : Functions Function"
    p[0] = p[1] + p[2]

def p_Functions_empty(p):
    "Functions : "
    p[0] = ""

def p_Function(p):
```

```
    "Function : DEF ID Instructions END"

    add_func(p[2], p)

    p[0] = f"{p[2]}:\n{p[3]}return\n"

# ------------------------------------------------------------------
#                          DECLARATIONS
# ------------------------------------------------------------------

def p_Declarations(p):
    "Declarations : Declarations Declaration"

    p[0] = p[1] + p[2]

def p_Declarations_empty(p):
    "Declarations : "

    p[0] = ""

def p_Declararation_exp(p):
    "Declaration : INT ID ATTR Exp SC"

    add_var(p[2], 1, 1, p)

    p[0] = p[4]

def p_Declararation_simple(p):
    "Declaration : INT ID SC"

    add_var(p[2], 1, 1, p)

    p[0] = "pushi 0\n"


def p_Declararation_array_num(p):
    "Declaration : INT ID LBRA NUM RBRA SC"

    col = int(p[4])
    add_var(p[2], col, 1, p)

    p[0] = f"pushn {col}\n"


def p_Declararation_matrix_num(p):
    "Declaration : INT ID LBRA NUM RBRA LBRA NUM RBRA SC"

    add_var(p[2], int(p[7]), int(p[4]), p)
```

```
    size = int(p[7]) * int(p[4])
    p[0] = f"pushn {size}\n"

# ------------------------------------------------------------------
#                          INSTRUCTIONS
# ------------------------------------------------------------------

def p_Instructions(p):
    "Instructions : Instructions Instruction"

    p[0] = p[1] + p[2]

def p_Instructions_empty(p):
    "Instructions : "

    p[0] = ""

# ------------------------------------------------------------------
#                         CALL FUNCTION
# ------------------------------------------------------------------

def p_Call(p):
    "Instruction : CALL ID SC"

    add_func_called(p[2], p)

    p[0] = f"pusha {p[2]}\ncall\n"

# ------------------------------------------------------------------
#                             INPUT
# ------------------------------------------------------------------

def p_Read_id(p):
    "Instruction : INPUT ID SC"

    res = get_index(p[2], p)

    (_, _, offset) = res
    p[0] = f'''pushs ">> "\nwrites\nread\natoi\nstoreg {offset}\n'''

def p_Read_array(p):
    "Instruction : INPUT ID LBRA Exp RBRA SC"

    res = get_index(p[2], p)

    (_, _, offset) = res
    p[0] = f'''pushs ">> "\nwrites\npushgp\npushi {offset}\npadd\n{p[4]}read\natoi\nstoren\n'''
```

```
def p_Read_matrix(p):
    "Instruction : INPUT ID LBRA Exp RBRA LBRA Exp RBRA SC"

    res = get_index(p[2], p)

    (col, _, offset) = res
    p[0] = f'''pushs ">> "\nwrites\npushgp\npushi {offset}\npadd\n{p[4]}pushi {col}\nmul\n{p[7]}a


# ------------------------------------------------------------------
#                           PRINT
# ------------------------------------------------------------------

def p_Print(p):
    "Instruction : PRINT Line SC"
    p[0] = p[2]

def p_Line_String(p):
    "Line : STRING"

    p[0] = f'''pushs {p[1]}\nwrites\npushs "\\n"\nwrites\n'''

def p_Line_Log(p):
    "Line : Log"

    p[0] = f'''{p[1]}writei\npushs "\\n"\nwrites\n'''

# ------------------------------------------------------------------
#                           ATTRIBUTE
# ------------------------------------------------------------------

def p_Attribure(p):
    "Instruction : ID ATTR Exp SC"

    res = get_index(p[1], p)

    (_, _, offset) = res
    p[0] = f"{p[3]}storeg {offset}\n"

def p_Attribure_array(p):
    "Instruction : ID LBRA Exp RBRA ATTR Exp SC"

    res = get_index(p[1], p)

    (_, _, offset) = res

    p[0] = f"pushgp\npushi {offset}\npadd\n{p[3]}{p[6]}storen\n"
```

```
def p_Attribure_matrix(p):
    "Instruction : ID LBRA Exp RBRA LBRA Exp RBRA ATTR Exp SC"

    res = get_index(p[1], p)

    (col, _, offset) = res

    p[0] = f"pushgp\npushi {offset}\npadd\n{p[3]}pushi {col}\nmul\n{p[6]}add\n{p[9]}storen\n"
# ------------------------------------------------------------------
#                               IF
# ------------------------------------------------------------------

def p_Condition(p):
    "Instruction : IF LPAR Log RPAR THEN Instructions ELSE Instructions END"
    p[0] = f"{p[3]}jz else{p.parser.ifs}\n{p[6]}jump endif{p.parser.ifs}\nelse{p.parser.ifs}:\n{
    p.parser.ifs += 1

def p_Condition_simple(p):
    "Instruction : IF LPAR Log RPAR THEN Instructions END"
    p[0] = f"{p[3]}jz endif{p.parser.ifs}\n{p[6]}\nendif{p.parser.ifs}:\n"
    p.parser.ifs += 1

# ------------------------------------------------------------------
#                            CYCLE
# ------------------------------------------------------------------

def p_Cycle_Repeat_Until(p):
    "Instruction : REPEAT Instructions UNTIL LPAR Log RPAR END"
    p[0] = f"cycle{p.parser.cycles}:\n{p[2]}{p[5]}jz cycle{p.parser.cycles}\n"
    p.parser.cycles += 1

def p_Cycle_While_Do(p):
    "Instruction : WHILE LPAR Log RPAR DO Instructions END"
    p[0] = f"cycle{p.parser.cycles}:\n{p[3]}jz endcycle{p.parser.cycles}\n{p[6]}jump cycle{p.pars
    p.parser.cycles += 1

def p_Cycle_For_Do(p):
    "Instruction : FOR LPAR ID ATTR Exp SC Log SC ID ATTR Exp RPAR DO Instructions END"

    res1 = get_index(p[3], p)
    (_, _, offset1) = res1

    res2 = get_index(p[9], p)
    (_, _, offset2) = res2

    p[0] = f"{p[5]}storeg {offset1}\ncycle{p.parser.cycles}:\n{p[7]}jz endcycle{p.parser.cycles}
    p.parser.cycles += 1
```

```
# ------------------------------------------------------------
#                           OPERATIONS
# ------------------------------------------------------------

def p_Log_and(p):
    "Log : Log AND Rel"
    p[0] = f"{p[1]}{p[3]}mul\n"

def p_Log_or(p):
    "Log : Log OR Rel"
    p[0] = f"{p[1]}{p[3]}add\n{p[1]}{p[3]}mul\nsub\n"

def p_Log_not(p):
    "Log : NOT Rel"
    p[0] = "{p[2]}not\n"

def p_Log_Rel(p):
    "Log : Rel"
    p[0] = p[1]

def p_Rel_g(p):
    "Rel : Rel G Rel2"
    p[0] = f"{p[1]}{p[3]}sup\n"

def p_Rel_ge(p):
    "Rel : Rel GE Rel2"
    p[0] = f"{p[1]}{p[3]}supeq\n"

def p_Rel_l(p):
    "Rel : Rel L Rel2"
    p[0] = f"{p[1]}{p[3]}inf\n"

def p_Rel_le(p):
    "Rel : Rel LE Rel2"
    p[0] = f"{p[1]}{p[3]}infeq\n"

def p_Rel_rel2(p):
    "Rel : Rel2"
    p[0] = p[1]

def p_Rel2_eq(p):
    "Rel2 : Rel2 EQ Exp"
    p[0] = f"{p[1]}{p[3]}equal\n"

def p_Rel2_ne(p):
    "Rel2 : Rel2 NE Exp"
    p[0] = f"{p[1]}{p[3]}equal\nnot\n"
```

```
def p_Rel2_lexp(p):
    "Rel2 : Exp"
    p[0] = p[1]

def p_Exp_add(p):
    "Exp : Exp PLUS Term"
    p[0] = f"{p[1]}{p[3]}add\n"

def p_Exp_sub(p):
    "Exp : Exp MINUS Term"
    p[0] = f"{p[1]}{p[3]}sub\n"

def p_Exp_term(p):
    "Exp : Term"
    p[0] = p[1]

def p_Term_mult(p):
    "Term : Term MUL Factor"
    p[0] = f"{p[1]}{p[3]}mul\n"

def p_Term_div(p):
    "Term : Term DIV Factor"
    p[0] = f"{p[1]}{p[3]}div\n"

def p_Term_mod(p):
    "Term : Term MOD Factor"
    p[0] = f"{p[1]}{p[3]}mod\n"

def p_Term_factor(p):
    "Term : Factor"
    p[0] = p[1]

def p_Factor_par(p):
    "Factor : LPAR Log RPAR"
    p[0] = p[2]

def p_Factor_num(p):
    "Factor : NUM"
    p[0] = f"pushi {p[1]}\n"

def p_Factor_id(p):
    "Factor : ID"
    res = get_index(p[1], p)

    (_, _, offset) = res
    p[0] = f"pushg {offset}\n"
```

```
def p_Factor_array(p):
    "Factor : ID LBRA Exp RBRA"

    res = get_index(p[1], p)

    (_, _, offset) = res
    p[0] = f"pushgp\npushi {offset}\npadd\n{p[3]}loadn\n"

def p_Factor_matrix(p):
    "Factor : ID LBRA Exp RBRA LBRA Exp RBRA"

    res = get_index(p[1], p)

    (col, _, offset) = res
    p[0] = f"pushgp\npushi {offset}\npadd\n{p[3]}pushi {col}\nmul\n{p[6]}add\nloadn\n"

# ------------------------------------------------------------------
#                           OTHER
# ------------------------------------------------------------------

def p_error(p):
    print("\nCompilation error, aborting!")

def add_var(id, col, line, p):
    if id not in p.parser.var.keys():
        p.parser.var[id] = (col, line, p.parser.offset)
        p.parser.offset += col * line
    else:
        raise Exception

def get_index(id, p):
    if id in p.parser.var.keys():
        return p.parser.var[id]

def add_func(name, p):
    if name not in p.parser.funcs:
        p.parser.funcs.add(name)
    else:
        raise Exception

def add_func_called(name, p):
    if name not in p.parser.called_funcs:
        p.parser.called_funcs.add(name)

def check_called_funcs(parser):
    for called in parser.called_funcs:
        if called not in parser.funcs:
            raise Exception
```

```
# ------------------------------------------------------------
#                           RUN
# ------------------------------------------------------------

r = 1
while r:
    inFilePath = input("Code File Path >> ")

    try:
        fileIn = open(inFilePath, "r")
        r = 0
    except (FileNotFoundError, NotADirectoryError):
        print("Wrong File Path\n")

r = 1
while r:
    outFilePath = input("Output File Path >> ")

    if outFilePath != inFilePath:
        try:
            fileOut = open(outFilePath, "w")
            r = 0
        except (FileNotFoundError, NotADirectoryError):
            print("Wrong File Path\n")
    else:
        print("Wrong File Path\n")

parser = yacc.yacc()

parser.var = dict() # x => (col, line, offset)
parser.funcs = set()
parser.called_funcs = set()
parser.offset = 0
parser.ifs = 0
parser.cycles = 0

data = fileIn.read()

try:
    result = parser.parse(data)
    fileOut.write(result)
    check_called_funcs(parser)
except (TypeError, Exception):
    print("\nCompilation error, aborting!")
    os.remove(outFilePath)

fileIn.close()
```

```
fileOut.close()
```