

Processamento de Linguagens e Compiladores (3º ano LCC)

Trabalho Prático 2

(Gramáticas, Compiladores)

Grupo 17

Relatório de Desenvolvimento

José Pedro Gomes Ferreira
A91636

Pedro Paulo Costa Pereira
A88062

Tiago André Oliveira Leite
A91693

7 de Janeiro de 2022

Conteúdo

1	Introdução	3
2	Problema Proposto	4
2.1	Descrição	4
2.2	Requisitos	4
3	Concepção/desenho da Resolução	6
3.1	Organização e estrutura	6
3.2	GIC	6
3.3	Lexer	8
3.4	Parser e geração do código Assembly da VM	9
4	Demonstração de Funcionamento	10
4.1	Geração e execução de código Assembly	10
4.2	Teste 1	10
4.2.1	Conteúdo do ficheiro	10
4.2.2	Código assembly gerado	11
4.2.3	Execução da VM com o código gerado	16
4.3	Teste 2	16
4.3.1	Conteúdo do ficheiro	17
4.3.2	Código assembly gerado	17
4.3.3	Execução da VM com o código gerado	19
4.4	Teste 3	19
4.4.1	Conteúdo do ficheiro	20
4.4.2	Código assembly gerado	20
4.4.3	Execução da VM com o código gerado	22
4.5	Teste 4	23
4.5.1	Conteúdo do ficheiro	23
4.5.2	Código assembly gerado	24
4.5.3	Execução da VM com o código gerado	25
4.6	Teste 5	26
4.6.1	Conteúdo do ficheiro	26
4.6.2	Código assembly gerado	26

4.6.3	Execução da VM com o código gerado	28
4.7	Teste 6	29
4.7.1	Conteúdo do ficheiro	29
4.7.2	Código assembly gerado	29
4.7.3	Execução da VM com o código gerado	30
4.8	Teste 7	30
4.8.1	Conteúdo do ficheiro	31
4.8.2	Código assembly gerado	32
4.8.3	Execução da VM com o código gerado	36
5	Conclusão	38
A	Código do Programa	39

Capítulo 1

Introdução

Este documento tem como objetivo explicar a solução que implementamos para a resolução do problema proposto no âmbito da unidade curricular de Processamento de Linguagens e Compiladores.

O problema consiste em implementar uma linguagem de programação imperativa simples, com regras sintáticas definidas pelo grupo.

Para o desenvolvimento da nossa linguagem tivemos que definir uma gramática independente de contexto, **GIC**, e desenvolver um compilador que gera **pseudo-código Assembly** para uma Máquina Virtual VM, que nos foi fornecida.

O trabalho foi desenvolvido com recurso aos módulos 'Yacc/ Lex' do 'PLY/Python'.

Esperamos que o resultado final cumpra todos os requisitos.

Capítulo 2

Problema Proposto

2.1 Descrição

Pretende-se que comece por definir uma linguagem de programação imperativa simples, a seu gosto. Apenas deve ter em consideração que essa linguagem terá de permitir:

- *declarar* variáveis atômicas 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**.

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

- *declarar e manusear* variáveis estruturadas do tipo *array* (a 1 ou 2 dimensões) de inteiros, em relação aos quais é apenas permitida a operação de indexação (índice inteiro);
- *definir e invocar subprogramas* sem parâmetros mas que possam retornar um resultado do tipo inteiro.

2.2 Requisitos

- Utilização da linguagem Python.
- Resolver o problema recurso aos módulos 'Yacc/ Lex' do 'PLY/Python'.
- Gerar **pseudo-código**, **Assembly** da Máquina Virtual VM fornecida.
- Preparar um conjunto de testes de modo a ver o código Assembly gerado bem como o programa a correr na máquina virtual VM. Este conjunto terá de conter, no mínimo, os 4 primeiros exemplos abaixo e um dos 2 últimos conforme a escolha de funcionalidades da linguagem:
 - 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) números e calcular e imprimir o seu produtório;
- contar e imprimir os números ímpares de uma sequência de números naturais;
- ler e armazenar N números num array; imprimir os valores por ordem inversa;
- invocar e usar num programa ou uma função 'potencia()', que começa por ler do input a base B e o expoente E e retorna o valor B^E .

Capítulo 3

Concepção/desenho da Resolução

3.1 Organização e estrutura

O nosso trabalho pode ser dividido em 4 partes:

- Definição da **GIC** que define a estrutura sintática da nossa linguagem.
- Construção do analisador léxico, **lexer**.
- Construção do analisador sintático, **parser**.
- Conversão das instruções para código **Assembly** da VM.

Todas as funcionalidades descritas neste capítulo podem ser encontradas no anexo A do documento.

No nosso trabalho optamos por implementar a funcionalidade de `textit` declarar e manusear variáveis estruturadas do tipo *array* (a 1 ou 2 dimensões) de inteiros, em relação aos quais é apenas permitida a operação de indexação (índice inteiro).

3.2 GIC

A nossa linguagem é gerada pela seguinte gramática independente de contexto:

```
program      : MAIN LCURLY body RCURLY

body         : declarations instructions

declarations :
              | declaration declarations

declaration  : type VAR SEMICOLON
              | type LBRACKET NUM RBRACKET VAR SEMICOLON
              | type LBRACKET NUM RBRACKET LBRACKET NUM RBRACKET VAR SEMICOLON

type         : INT
```

```

| FLOAT

instructions :
| instruction instructions

instruction : atributions SEMICOLON
| WHILE LPAREN condition RPAREN LCURLY instructions RCURLY
| FOR LPAREN atributions SEMICOLON condition SEMICOLON atributions RPAREN LCURLY
instructions RCURLY
| IF LPAREN condition RPAREN LCURLY instructions RCURLY
| IF LPAREN condition RPAREN LCURLY instructions RCURLY ELSE LCURLY
instructions RCURLY
| SCAN LPAREN variable RPAREN SEMICOLON
| PRINT LPAREN variable RPAREN SEMICOLON
| PRINTLN LPAREN variable RPAREN SEMICOLON
| PRINT LPAREN STRING RPAREN SEMICOLON
| PRINTLN LPAREN STRING RPAREN SEMICOLON

atributions :
| atribution
| atribution COMMA atributions

atribution : variable EQUAL expression
| variable EQUAL condition

variable : VAR
| VAR LBRACKET expression RBRACKET
| VAR LBRACKET expression RBRACKET LBRACKET expression RBRACKET

expression : variable
| NUM
| REAL
| LPAREN expression RPAREN
| expression PLUS expression
| expression MINUS expression
| expression MUL expression
| expression DIV expression
| expression MOD expression

condition : expression EQEQ expression
| expression DIFF expression
| expression GREATER expression
| expression LESSER expression
| expression GREATEQ expression
| expression LESSEQ expression
| NUM
| REAL
| variable

```


3.3 Lexer

O analisador léxico, **lexer**, é o responsável por 'capturar' os símbolos terminais, **tokens**, da nossa linguagem através de expressões regulares. Para a implementação do analisador léxico utilizamos o módulo 'Lex' do 'PLY/Python'.

Os **tokens** e respectivas expressões regulares da nossa linguagem são os seguintes:

```
SEMICOLON : ';'
COMMA     : ','
LCURLY    : '{'
RCURLY    : '}'
LPAREN    : '('
RPAREN    : ')'
LBRACKET  : '['
RBRACKET  : ']'
FLOAT     : 'float'
INT       : 'int'
MAIN      : 'main'
WHILE     : 'while'
FOR       : 'for'
IF        : 'if'
ELSE      : 'else'
PRINTLN   : 'println'
PRINT     : 'print'
SCAN      : 'scan'
STRING    : '"'([^\"]|\\n)*'"
REAL      : '-?([1-9][0-9]*\\.[0-9]+|0\\.[0-9]+)'
NUM       : '-?\\d+'
EQUAL     : '\\='
DIFF      : '\\!='
GREATEREQ : '\\>='
LESSEQ    : '\\<='
GREATER   : '\\>'
LESSER    : '\\<'
EQEQ      : '\\=='
PLUS      : '\\+'
MINUS     : '\\-'
MUL       : '\\*'
DIV       : '\\/'
MOD       : '\\%'
VAR       : '\\w+'
```

A implementação do analisador léxico pode ser encontrada no anexo A do documento.

3.4 Parser e geração do código Assembly da VM

O analisador sintático, **parser**, é responsável por verificar se o código que foi escrito na nossa linguagem está sintaticamente correto, isto é, se respeita as regras gramaticais definidas.

Caso não haja erros sintáticos o converte o código da nossa linguagem em código **Assembly** da máquina virtual. Se houver erros, é mostrado ao utilizador uma mensagem de erro sintático.

A implementação do analisador sintático pode ser encontrada no anexo A do documento.

Capítulo 4

Demonstração de Funcionamento

4.1 Geração e execução de código Assembly

Para utilizar a nossa linguagem, o utilizador tem que:

1. Escrever e guardar as instruções num ficheiro de texto de acordo com as regras grámaticais da linguagem.
2. Colocar o ficheiro na mesma diretoria dos ficheiros `lexer.py`, `parser.py`, `vms` e `vmsGTKAux`.
3. Executar um dos seguintes comandos:

```
>> python3 parser.py <ficheiro de input>
>> python3 parser.py <ficheiro de input> <ficheiro de output>
```

Nota: Caso o utilizador opte pelo primeiro comando é criado um ficheiro denominado `a.vm` onde será colocado o código Assembly gerado.

4. Executar o comando:

```
>> ./vms <ficheiro gerado no comando anterior>
```

4.2 Teste 1

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

Ficheiro de input: 'quadrado.txt'.

4.2.1 Conteúdo do ficheiro

```
main
{
    float[4] lados;
    int[6] a;
    int i;
    i = 4;
```

```

while(i){
    print("Digite lado: ");
    scan(lados[4 - i]);
    i = i - 1;
}

a[0] = lados[0] == lados[1];
a[1] = lados[0] == lados[2];
a[2] = lados[0] == lados[3];
a[3] = lados[1] == lados[2];
a[4] = lados[1] == lados[3];
a[5] = lados[2] == lados[3];

i = a[0]*a[1]*a[2]*a[3]*a[4]*a[5];

if(i)
{
    if(lados[0])
    {
        println("Podem ser os lados de um quadrado!");
    }

    else
    {
        println("Não podem ser os lados de um quadrado!");
    }
}

else
{
    println("Não podem ser os lados de um quadrado!");
}
}

```

4.2.2 Código assembly gerado

```

PUSHN 4
PUSHN 6
PUSHI 0
START
PUSHGP
PUSHI 10
PUSHI 4
STOREN

```

```

B0:
PUSHGP
PUSHI 10
LOADN
PUSHI 0
SUP
JZ E0
PUSHS "Digite lado: "
WRITES
PUSHGP
PUSHI 0
PUSHI 4
PUSHGP
PUSHI 10
LOADN
SUB
ADD
READ
ATOF
STOREN
PUSHGP
PUSHI 10
PUSHGP
PUSHI 10
LOADN
PUSHI 1
SUB
STOREN
JUMP B0
E0:
PUSHGP
PUSHI 4
PUSHI 0
ADD
PUSHGP
PUSHI 0
PUSHI 0
ADD
LOADN
PUSHGP
PUSHI 0
PUSHI 1
ADD
LOADN
EQUAL
STOREN
PUSHGP
PUSHI 4

```

PUSHI 1
ADD
PUSHGP
PUSHI 0
PUSHI 0
ADD
LOADN
PUSHGP
PUSHI 0
PUSHI 2
ADD
LOADN
EQUAL
STOREN
PUSHGP
PUSHI 4
PUSHI 2
ADD
PUSHGP
PUSHI 0
PUSHI 0
ADD
LOADN
PUSHGP
PUSHI 0
PUSHI 3
ADD
LOADN
EQUAL
STOREN
PUSHGP
PUSHI 4
PUSHI 3
ADD
PUSHGP
PUSHI 0
PUSHI 1
ADD
LOADN
PUSHGP
PUSHI 0
PUSHI 2
ADD
LOADN
EQUAL
STOREN
PUSHGP
PUSHI 4

PUSHI 4
ADD
PUSHGP
PUSHI 0
PUSHI 1
ADD
LOADN
PUSHGP
PUSHI 0
PUSHI 3
ADD
LOADN
EQUAL
STOREN
PUSHGP
PUSHI 4
PUSHI 5
ADD
PUSHGP
PUSHI 0
PUSHI 2
ADD
LOADN
PUSHGP
PUSHI 0
PUSHI 3
ADD
LOADN
EQUAL
STOREN
PUSHGP
PUSHI 10
PUSHGP
PUSHI 4
PUSHI 0
ADD
LOADN
PUSHGP
PUSHI 4
PUSHI 1
ADD
LOADN
MUL
PUSHGP
PUSHI 4
PUSHI 2
ADD
LOADN

```

MUL
PUSHGP
PUSHI 4
PUSHI 3
ADD
LOADN
MUL
PUSHGP
PUSHI 4
PUSHI 4
ADD
LOADN
MUL
PUSHGP
PUSHI 4
PUSHI 5
ADD
LOADN
MUL
STOREN
PUSHGP
PUSHI 10
LOADN
PUSHI 0
SUP
JZ E2
PUSHGP
PUSHI 0
PUSHI 0
ADD
LOADN
PUSHF 0.0
FSUP
FTOI
PUSHI 0
SUP
JZ E1
PUSHS "Podem ser os lados de um quadrado!"
WRITES
PUSHS"\n"
WRITES
JUMP F1
E1:
PUSHS "Não podem ser os lados de um quadrado!"
WRITES
PUSHS"\n"
WRITES
F1:

```



```

JUMP F2
E2:
PUSHS "Não podem ser os lados de um quadrado!"
WRITES
PUSHS "\n"
WRITES
F2:
STOP

```

4.2.3 Execução da VM com o código gerado

```

>> ./vms a.vm
Digite lado: 1
Digite lado: 1
Digite lado: 1
Digite lado: 1
Podem ser os lados de um quadrado!
>>

```

```

>> ./vms a.vm
Digite lado: 0
Digite lado: 0
Digite lado: 0
Digite lado: 0
Não podem ser os lados de um quadrado!
>>

```

```

>> ./vms a.vm
Digite lado: 0.1
Digite lado: 0.1
Digite lado: 0.1
Digite lado: 0.1
Podem ser os lados de um quadrado!

```

```

>> ./vms a.vm
Digite lado: 4
Digite lado: 4
Digite lado: 6
Digite lado: 4
Não podem ser os lados de um quadrado!
>>

```

4.3 Teste 2

Ler um inteiro N, depois ler N números e escrever o menor deles.
Ficheiro de input: 'menor.txt'.

4.3.1 Conteúdo do ficheiro

```
main
{

    int menor;
    int N;
    int i;
    int aux;

    i = 0;

    print("Digite número: ");
    scan(menor);

    print("Digite quantos números quer ler: ");
    scan(N);

    while(i<N){
        print("Digite número: ");
        scan(aux);

        if(aux < menor){
            menor = aux;
        }
        i = i +1;
    }
    print("O menor número é: ");
    println(menor);
}
```

4.3.2 Código assembly gerado

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
PUSHGP
PUSHI 2
PUSHI 0
STOREN
PUSHS "Digite número: "
WRITES
PUSHGP
```

```

PUSHI 0
READ
ATOI
STOREN
PUSHS "Digite quantos números quer ler: "
WRITES
PUSHGP
PUSHI 1
READ
ATOI
STOREN
B1:
PUSHGP
PUSHI 2
LOADN
PUSHGP
PUSHI 1
LOADN
INF
JZ E1
PUSHS "Digite número: "
WRITES
PUSHGP
PUSHI 3
READ
ATOI
STOREN
PUSHGP
PUSHI 3
LOADN
PUSHGP
PUSHI 0
LOADN
INF
JZ E0
PUSHGP
PUSHI 0
PUSHGP
PUSHI 3
LOADN
STOREN
E0:
PUSHGP
PUSHI 2
PUSHGP
PUSHI 2
LOADN
PUSHI 1

```

```

ADD
STOREN
JUMP B1
E1:
PUSHS "O menor número é: "
WRITES
PUSHGP
PUSHI 0
LOADN
WRITEI
PUSHS "\n"
WRITES
STOP

```

4.3.3 Execução da VM com o código gerado

```

>> ./vms a.vm
Digite número: 5
Digite quantos números quer ler: 3
Digite número: -1
Digite número: 4
Digite número: 0
O menor número é: -1
>>

```

```

>> ./vms a.vm
Digite número: 2
Digite quantos números quer ler: 0
O menor número é: 2
>>

```

```

>> ./vms a.vm
Digite número: 1
Digite quantos números quer ler: 5
Digite número: 2
Digite número: 3
Digite número: 4
Digite número: 5
Digite número: 0
O menor número é: 0

```

4.4 Teste 3

Ler N (constante do programa) números e calcular e imprimir o seu produtório.
Ficheiro de input: 'produto.txt'.

4.4.1 Conteúdo do ficheiro

```
main
{

    int N;
    int r;
    int[5] a;
    int i;

    i = 0;
    N = 5;
    r = 1;

    while(i<N){
        print("Digite um número: ");
        scan(a[i]);
        i = i + 1;

    }
    i = 0;

    while(i<N){
        print(a[i]);
        print(" x ");
        r = r * a[i];
        i = i + 1;
    }

    print(" = ");
    println(r);

}
```

4.4.2 Código assembly gerado

```
PUSHI 0
PUSHI 0
PUSHN 5
PUSHI 0
START
PUSHGP
PUSHI 7
PUSHI 0
STOREN
PUSHGP
PUSHI 0
PUSHI 5
```

```
STOREN
PUSHGP
PUSHI 1
PUSHI 1
STOREN
B0:
PUSHGP
PUSHI 7
LOADN
PUSHGP
PUSHI 0
LOADN
INF
JZ E0
PUSHS "Digite um número: "
WRITES
PUSHGP
PUSHI 2
PUSHGP
PUSHI 7
LOADN
ADD
READ
ATOI
STOREN
PUSHGP
PUSHI 7
PUSHGP
PUSHI 7
LOADN
PUSHI 1
ADD
STOREN
JUMP B0
E0:
PUSHGP
PUSHI 7
PUSHI 0
STOREN
B1:
PUSHGP
PUSHI 7
LOADN
PUSHGP
PUSHI 0
LOADN
INF
JZ E1
```

```

PUSHGP
PUSHI 2
PUSHGP
PUSHI 7
LOADN
ADD
LOADN
WRITEI
PUSHS " x "
WRITES
PUSHGP
PUSHI 1
PUSHGP
PUSHI 1
LOADN
PUSHGP
PUSHI 2
PUSHGP
PUSHI 7
LOADN
ADD
LOADN
MUL
STOREN
PUSHGP
PUSHI 7
PUSHGP
PUSHI 7
LOADN
PUSHI 1
ADD
STOREN
JUMP B1
E1:
PUSHS " = "
WRITES
PUSHGP
PUSHI 1
LOADN
WRITEI
PUSHS "\n"
WRITES
STOP

```

4.4.3 Execução da VM com o código gerado

```

>> ./vms a.vm
Digite um número: 1

```

```

Digite um número: 2
Digite um número: 3
Digite um número: 4
Digite um número: 5
1 x 2 x 3 x 4 x 5 x  = 120
>>

```

```

>> ./vms a.vm
Digite um número: 0
Digite um número: 1
Digite um número: 2
Digite um número: 3
Digite um número: 4
0 x 1 x 2 x 3 x 4 x  = 0
>>

```

```

>> ./vms a.vm
Digite um número: 1.5
Digite um número: 1.5
Digite um número: 1
Digite um número: 2
Digite um número: 3
1 x 1 x 1 x 2 x 3 x  = 6
>>

```

4.5 Teste 4

Contar e imprimir os números ímpares de uma sequência de números naturais.
Ficheiro de input: 'impares.txt'.

4.5.1 Conteúdo do ficheiro

```

main
{

    int count;
    int aux;
    count = 0;
    aux = 1;
    println("Digite 0 para parar!");
    while(aux !=0){
        print("Digite número: ");
        scan(aux);
        if(aux %2 == 1){
            print(aux);
            println(" é impar!");
            count = count +1;
        }
    }
}

```



```

}
print("Foram lidos ");
print(count);
println(" números impares!");
}

```

4.5.2 Código assembly gerado

```

PUSHI 0
PUSHI 0
START
PUSHGP
PUSHI 0
PUSHI 0
STOREN
PUSHGP
PUSHI 1
PUSHI 1
STOREN
PUSHS "Digite 0 para parar!"
WRITES
PUSHS "\n"
WRITES
B1:
PUSHGP
PUSHI 1
LOADN
PUSHI 0
EQUAL
NOT
JZ E1
PUSHS "Digite número: "
WRITES
PUSHGP
PUSHI 1
READ
ATOI
STOREN
PUSHGP
PUSHI 1
LOADN
PUSHI 2
MOD
PUSHI 1
EQUAL
JZ E0
PUSHGP

```

```

PUSHI 1
LOADN
WRITEI
PUSHS " é impar!"
WRITES
PUSHS"\n"
WRITES
PUSHGP
PUSHI 0
PUSHGP
PUSHI 0
LOADN
PUSHI 1
ADD
STOREN
E0:
JUMP B1
E1:
PUSHS "Foram lidos "
WRITES
PUSHGP
PUSHI 0
LOADN
WRITEI
PUSHS " números impares!"
WRITES
PUSHS"\n"
WRITES
STOP

```

4.5.3 Execução da VM com o código gerado

```

>> ./vms a.vm
Digite 0 para parar!
Digite número: 1
1 é impar!
Digite número: 2
Digite número: 3
3 é impar!
Digite número: 4
Digite número: 5
5 é impar!
Digite número: 6
Digite número: 0
Foram lidos 3 números impares!
>>

```

4.6 Teste 5

Ler e armazenar N números num array. Imprimir os valores por ordem inversa.
Ficheiro de input: 'inversa.txt'.

4.6.1 Conteúdo do ficheiro

```
main
{
    int N;
    int i;
    int[10] a;
    N = 5;
    i = 0;

    print("Neste programa vamos digitar ");
    print(N);
    println(" números e imprimi-los por ordem inversa.");

    while(i<N){
        print("Digite número: ");
        scan(a[i]);
        i = i +1;
    }

    i = N - 1;

    while(i>=0){
        print(a[i]);
        print(" ");
        i = i - 1;
    }

    println("");
}
```

4.6.2 Código assembly gerado

```
PUSHI 0
PUSHI 0
PUSHN 10
START
PUSHGP
PUSHI 0
PUSHI 5
STOREN
PUSHGP
```

```

PUSHI 1
PUSHI 0
STOREN
PUSHS "Neste programa vamos digitar "
WRITES
PUSHGP
PUSHI 0
LOADN
WRITEI
PUSHS " números e imprimi-los por ordem inversa."
WRITES
PUSHS "\n"
WRITES
BO:
PUSHGP
PUSHI 1
LOADN
PUSHGP
PUSHI 0
LOADN
INF
JZ EO
PUSHS "Digite número: "
WRITES
PUSHGP
PUSHI 2
PUSHGP
PUSHI 1
LOADN
ADD
READ
ATOI
STOREN
PUSHGP
PUSHI 1
PUSHGP
PUSHI 1
LOADN
PUSHI 1
ADD
STOREN
JUMP BO
EO:
PUSHGP
PUSHI 1
PUSHGP
PUSHI 0
LOADN

```

```

PUSHI 1
SUB
STOREN
B1:
PUSHGP
PUSHI 1
LOADN
PUSHI 0
SUPEQ
JZ E1
PUSHGP
PUSHI 2
PUSHGP
PUSHI 1
LOADN
ADD
LOADN
WRITEI
PUSHS " "
WRITES
PUSHGP
PUSHI 1
PUSHGP
PUSHI 1
LOADN
PUSHI 1
SUB
STOREN
JUMP B1
E1:
PUSHS ""
WRITES
PUSHS "\n"
WRITES
STOP

```

4.6.3 Execução da VM com o código gerado

```

>> ./vms a.vm
Neste programa vamos digitar 5 números e imprimi-los por ordem inversa.
Digite número: 1
Digite número: 2
Digite número: 3
Digite número: 4
Digite número: 5
5 4 3 2 1
>>

```

```
>> ./vms a.vm
Neste programa vamos digitar 5 números e imprimi-los por ordem inversa.
Digite número: 3
Digite número: 4
Digite número: 1
Digite número: 7
Digite número: 6
6 7 1 4 3
>>
```

4.7 Teste 6

Execução de operações aritméticas.
Ficheiro de input: 'calcul0.txt'.

4.7.1 Conteúdo do ficheiro

```
main
{
    int a;

    a = 2 * 3 + 4 * 5 - 1;

    print("a = 2 * 3 + 4 * 5 - 1 = ");
    println(a);

    a = (1 + a) * a;

    print("(1 + a) * a = ");
    println(a);
}
```

4.7.2 Código assembly gerado

```
PUSHI 0
START
PUSHGP
PUSHI 0
PUSHI 2
PUSHI 3
MUL
PUSHI 4
PUSHI 5
MUL
```

```

ADD
PUSHI 1
SUB
STOREN
PUSHS "a = 2 * 3 + 4 * 5 - 1 = "
WRITES
PUSHGP
PUSHI 0
LOADN
WRITEI
PUSHS "\n"
WRITES
PUSHGP
PUSHI 0
PUSHI 1
PUSHGP
PUSHI 0
LOADN
ADD
PUSHGP
PUSHI 0
LOADN
MUL
STOREN
PUSHS "(1 + a) * a = "
WRITES
PUSHGP
PUSHI 0
LOADN
WRITEI
PUSHS "\n"
WRITES
STOP

```

4.7.3 Execução da VM com o código gerado

```

>> ./vms a.vm
a = 2 * 3 + 4 * 5 - 1 = 25
(1 + a) * a = 650
>>

```

4.8 Teste 7

Ordenação de um array.

Ficheiro de input: 'ordena.txt'.

4.8.1 Conteúdo do ficheiro

```
main
{
    int N;
    int i;
    int j;
    int menor;
    int aux;
    int[5] a;

    N = 5;

    print("Neste programa vamos digitar ");
    print(N);
    println(" números e imprimi-los por ordem crescente.");

    for(i=0;i<N;i=i+1){
        print("Digite número: ");
        scan(a[i]);
    }

    for(i=0;i<N;i=i+1)
    {
        menor = i;

        for(j=i+1;j<N;j=j+1)
        {
            if(a[j]<a[menor])
            {
                menor = j;
            }
        }

        if(menor !=i)
        {
            aux = a[i];
            a[i] = a[menor];
            a[menor] = aux;
        }
    }
}
```



```

    for(i=0;i<N; i=i+1)
    {
        print(a[i]);
        print(" ");
    }

    println("");
}

```

4.8.2 Código assembly gerado

```

PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
PUSHN 5
START
PUSHGP
PUSHI 0
PUSHI 5
STOREN
PUSHS "Neste programa vamos digitar "
WRITES
PUSHGP
PUSHI 0
LOADN
WRITEI
PUSHS " números e imprimi-los por ordem crescente."
WRITES
PUSHS "\n"
WRITES
PUSHGP
PUSHI 1
PUSHI 0
STOREN
B0:
PUSHGP
PUSHI 1
LOADN
PUSHGP
PUSHI 0
LOADN
INF
JZ E0

```

```
PUSHS "Digite número: "  
WRITES  
PUSHGP  
PUSHI 5  
PUSHGP  
PUSHI 1  
LOADN  
ADD  
READ  
ATOI  
STOREN  
PUSHGP  
PUSHI 1  
PUSHGP  
PUSHI 1  
LOADN  
PUSHI 1  
ADD  
STOREN  
JUMP B0  
EO:  
PUSHGP  
PUSHI 1  
PUSHI 0  
STOREN  
B4:  
PUSHGP  
PUSHI 1  
LOADN  
PUSHGP  
PUSHI 0  
LOADN  
INF  
JZ E4  
PUSHGP  
PUSHI 3  
PUSHGP  
PUSHI 1  
LOADN  
STOREN  
PUSHGP  
PUSHI 2  
PUSHGP  
PUSHI 1  
LOADN  
PUSHI 1  
ADD  
STOREN
```

B2:
PUSHGP
PUSHI 2
LOADN
PUSHGP
PUSHI 0
LOADN
INF
JZ E2
PUSHGP
PUSHI 5
PUSHGP
PUSHI 2
LOADN
ADD
LOADN
PUSHGP
PUSHI 5
PUSHGP
PUSHI 3
LOADN
ADD
LOADN
INF
JZ E1
PUSHGP
PUSHI 3
PUSHGP
PUSHI 2
LOADN
STOREN
E1:
PUSHGP
PUSHI 2
PUSHGP
PUSHI 2
LOADN
PUSHI 1
ADD
STOREN
JUMP B2
E2:
PUSHGP
PUSHI 3
LOADN
PUSHGP
PUSHI 1
LOADN

EQUAL
NOT
JZ E3
PUSHGP
PUSHI 4
PUSHGP
PUSHI 5
PUSHGP
PUSHI 1
LOADN
ADD
LOADN
STOREN
PUSHGP
PUSHI 5
PUSHGP
PUSHI 1
LOADN
ADD
PUSHGP
PUSHI 5
PUSHGP
PUSHI 3
LOADN
ADD
LOADN
STOREN
PUSHGP
PUSHI 5
PUSHGP
PUSHI 3
LOADN
ADD
PUSHGP
PUSHI 4
LOADN
STOREN
E3:
PUSHGP
PUSHI 1
PUSHGP
PUSHI 1
LOADN
PUSHI 1
ADD
STOREN
JUMP B4
E4:

```

PUSHGP
PUSHI 1
PUSHI 0
STOREN
B5:
PUSHGP
PUSHI 1
LOADN
PUSHGP
PUSHI 0
LOADN
INF
JZ E5
PUSHGP
PUSHI 5
PUSHGP
PUSHI 1
LOADN
ADD
LOADN
WRITEI
PUSHS " "
WRITES
PUSHGP
PUSHI 1
PUSHGP
PUSHI 1
LOADN
PUSHI 1
ADD
STOREN
JUMP B5
E5:
PUSHS ""
WRITES
PUSHS "\n"
WRITES
STOP

```

4.8.3 Execução da VM com o código gerado

```

>> ./vms a.vm
Neste programa vamos digitar 5 números e imprimi-los por ordem crescente.
Digite número: 1
Digite número: 2
Digite número: 3
Digite número: 4
Digite número: 5

```

```
1 2 3 4 5
```

```
>>
```

```
>> ./vms a.vm
```

Neste programa vamos digitar 5 números e imprimi-los por ordem crescente.

Digite número: 5

Digite número: 4

Digite número: 3

Digite número: 2

Digite número: 1

```
1 2 3 4 5
```

```
>>
```

```
>> ./vms a.vm
```

Neste programa vamos digitar 5 números e imprimi-los por ordem crescente.

Digite número: -1

Digite número: -5

Digite número: 3

Digite número: 0

Digite número: 3

```
-5 -1 0 3 3
```

```
>>
```

4.9 Teste 8

Execução de operações aritméticas.

Ficheiro de input: 'erro1.txt'.

4.9.1 Conteúdo do ficheiro

```
main
```

```
{
```

```
    int a;
```

```
    int a;
```

```
}
```

4.9.2 Código assembly gerado

```
PUSHI 0
```

```
ERR "multipla declaração da variavel a\n"
```

```
STOP
```

```
START
```

```
STOP
```

4.9.3 Execução da VM com o código gerado

```
>> ./vms a.vm
```

```
multiplica declaração da variavel a
>>
```

4.10 Teste 9

Execução de operações aritméticas.
Ficheiro de input: 'erro2.txt'.

4.10.1 Conteúdo do ficheiro

```
main
{
    int a;
    a = b;
}
```

4.10.2 Código assembly gerado

```
PUSHI 0
START
PUSHGP
PUSHI 0
ERR "segmentation fault\n"
STOP
STOREN
STOP
```

4.10.3 Execução da VM com o código gerado

```
>> ./vms a.vm
segmentation fault
>>
```

Capítulo 5

Conclusão

Com o projeto concluído esperamos ter cumprido todos os requisitos que nos foram propostos.

O facto de produzirmos a nossa própria linguagem tornou a experiencia bastante interessante, apesar de a sintaxe escolhida para a linguagem desenvolvida se aproximar muito da linguagem C.

Achamos que há aspetos que poderiam ser melhorados, como por exemplo a implementação de funções com e sem argumentos. Também na parte de atribuição de valores as variaveis há uma pequena falha que permite ao utilizador por exemplo fazer:

```
a = 1,;
```

No entanto se esta fosse removida, nao poderiamos por exemplo fazer a seguinte instrução:

```
a = 1,b = 2;
```

E os ciclos *for* teriam que ficar do genero:

```
for(i=0;i<N;i=i+i;){}
```

Ou seja, a operação de incremento do 'i' teria que ter um ';' desnecessário.

Contudo, como temos a liberdade de definir a sintaxe da linguagem, podemos dizer que isto nao é um erro mas sim uma opção.

Por fim, tal como ja tinha sido mencionado no relatório do primeiro projeto, todos concordamos que o facto de o projecto ter sido desenvolvido na linguagem 'Python' e, neste caso, com recurso aos módulos 'Yacc/Lex' do 'PLY/Python', facilitou bastante o seu desenvolvimento.

Apêndice A

Código do Programa

Ficheiro lexer.py

```
import ply.lex as lex
import sys

tokens = ('LCURLY', 'RCURLY', 'LPAREN', 'RPAREN', 'LBRACKET', 'RBRACKET', 'NUM', 'REAL',
          'VAR', 'FLOAT', 'INT', 'SEMICOLON', 'COMMA', 'MAIN', 'WHILE', 'FOR', 'IF', 'ELSE',
          'STRING', 'EQUAL', 'PLUS', 'MINUS', 'MUL', 'DIV', 'MOD', 'EQQ', 'DIFF', 'GREATER',
          'LESSER', 'GREATERQ', 'LESSEQ', 'SCAN', 'PRINT', 'PRINTLN')

def t_SEMICOLON(t):
    r';'
    return t

def t_COMMA(t):
    r','
    return t

def t_LCURLY(t):
    r'\{'
    return t

def t_RCURLY(t):
    r'\}'
    return t

def t_LPAREN(t):
    r'\('
    return t

def t_RPAREN(t):
    r'\)'
    return t

def t_LBRACKET(t):
    r'\['
    return t

def t_RBRACKET(t):
    r'\]'
    return t

def t_FLOAT(t):
    r'float'
    return t

def t_INT(t):
    r'int'
    return t
```

```

def t_MAIN(t):
    r'main'
    return t

def t_WHILE(t):
    r'while'
    return t

def t_FOR(t):
    r'for'
    return t

def t_IF(t):
    r'if'
    return t

def t_ELSE(t):
    r'else'
    return t

def t_PRINTLN(t):
    r'println'
    return t

def t_PRINT(t):
    r'print'
    return t

def t_SCAN(t):
    r'scan'
    return t

def t_STRING(t):
    r'"([^\"]|\\n)*"'
    return t

def t_REAL(t):
    r'-?([1-9][0-9]*\.[0-9]+|0\.[0-9]+)'
    return t

def t_NUM(t):
    r'-?\d+'
    t.value = int(t.value)
    return t

def t_EQEQ(t):
    r'\==\'
    return t

def t_DIFF(t):
    r'\!=\'
    return t

def t_GRETEQ(t):
    r'\>=\'
    return t

def t_LESSEQ(t):
    r'\<=\'
    return t

def t_GREATER(t):
    r'\>'
    return t

def t_LESSER(t):
    r'\<'

```

```

    return t

def t_EQUAL(t):
    r'\='
    return t

def t_PLUS(t):
    r'\+'
    return t

def t_MINUS(t):
    r'\-'
    return t

def t_MUL(t):
    r'\*'
    return t

def t_DIV(t):
    r'\/'
    return t

def t_MOD(t):
    r'\%'
    return t

def t_VAR(t):
    r'\w+'
    return t

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

t_ignore = ' \r\n\t'

lexer = lex.lex()

```

Ficheiro parser.py

```

import ply.yacc as yacc
import sys
import os.path
from lexer import tokens

precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'MUL', 'DIV'),
)

def var_new(v):
    if v[0] in parser.tab_id or v[1] == 0 or v[2] == 0:
        return -1
    else:
        parser.tab_id[v[0]] = (parser.prox_address, v[1], v[2], v[3])
        parser.prox_address += v[1]*v[2]
        return 0

def var_address_base(v):
    if v in parser.tab_id:
        return parser.tab_id[v][0]
    else:
        return -1

def var_num_columns(v):

```

```

if v in parser.tab_id:
    return parser.tab_id[v][1]
else:
    return -1

def var_num_lines(v):
    if v in parser.tab_id:
        return parser.tab_id[v][2]
    else:
        return -1

def var_size(v):
    if v in parser.tab_id:
        return parser.tab_id[v][1] * parser.tab_id[v][2]
    else:
        return 0

def var_type(v):
    if v in parser.tab_id:
        return parser.tab_id[v][3]
    else:
        return None

def p_program(p):
    """
    program : MAIN LCURLY body RCURLY
    """
    fp.write(p[3])
    print(p[3])

def p_body(p):
    """
    body : declarations instructions
    """
    p[0] = p[1] + 'START\n' + p[2] + 'STOP'

def p_declarations_empty(p):
    """
    declarations :
    """
    p[0] = ""

def p_declarations(p):
    """
    declarations : declaration declarations
    """
    p[0] = p[1] + p[2]

def p_declaration_single(p):
    """
    declaration : type VAR SEMICOLON
    """
    status = var_new((p[2],1,1,p[1]))
    if status == -1:
        p[0] = f'ERR \"múltipla declaração da variavel {p[2]}\n\n\"nSTOP\n'
    else:
        p[0] = 'PUSHI 0\n'

def p_declaration_array(p):

```

```

"""
declaration : type LBRACKET NUM RBRACKET VAR SEMICOLON
"""
status = var_new((p[5],p[3],1,p[1]))
if status == -1:
    p[0] = f'ERR \"múltipla declaração da variavel {p[2]}\n\"\\nSTOP\n'
else:
    p[0] = f'PUSHN {var_size(p[5])}\n'

def p_declaration_biarray(p):
    """
    declaration : type LBRACKET NUM RBRACKET LBRACKET NUM RBRACKET VAR SEMICOLON
    """
    status = var_new((p[8],p[6],p[3],p[1]))
    if status == -1:
        p[0] = f'ERR \"múltipla declaração da variavel {p[2]}\n\"\\nSTOP\n'
    else:
        p[0] = f'PUSHN {var_size(p[8])}\n'

def p_type_int(p):
    """
    type : INT
    """
    p[0] = f'{p[1]}'

def p_type_float(p):
    """
    type : FLOAT
    """
    p[0] = f'{p[1]}'

def p_variable_single(p):
    """
    variable : VAR
    """
    p[0] = ('PUSHGP\n' + f'PUSHI {var_address_base(p[1])}\n',var_type(p[1]),var_size(p[1]))

def p_variable_index_expression(p):
    """
    variable : VAR LBRACKET expression RBRACKET
    """
    p[0] = ('PUSHGP\n' + f'PUSHI {var_address_base(p[1])}\n' + p[3][0] +
        'ADD\n',var_type(p[1]),var_size(p[1]))

def p_variable_index_expression_expression(p):
    """
    variable : VAR LBRACKET expression RBRACKET LBRACKET expression RBRACKET
    """
    p[0] = ('PUSHGP\n' + f'PUSHI {var_address_base(p[1])}\n' + f'PUSHI {var_num_columns(p[1])}\n'
        + p[3][0] + 'MUL\n' + 'ADD\n' + p[6][0] + 'ADD\n',var_type(p[1]),var_size(p[1]))

def p_instructions_empty(p):
    """
    instructions :
    """
    p[0] = ""

def p_instructions(p):
    """

```

```

instructions : instruction instructions
"""
p[0] = p[1] + p[2]

def p_instruction(p):
    """
    instruction : atributions SEMICOLON
    """
    p[0] = p[1]

def p_atributions_empty(p):
    """
    atributions :
    """
    p[0] = ""

def p_atributions_single(p):
    """
    atributions : attribution
    """
    p[0] = p[1]

def p_atributions_multiple(p):
    """
    atributions : attribution COMMA atributions
    """
    p[0] = p[1] + p[3]

def p_instruction_attribution_expression(p):
    """
    attribution : variable EQUAL expression
    """
    if p[1][1] == None:
        p[0] = f'ERR \segmentation fault\\n\\nSTOP\n'
    elif p[1][1] == 'int' and p[3][1] == 'float':
        p[0] = p[1][0] + p[3][0] + 'FTOI\n' + 'STOREN\n'
    elif p[1][1] == 'float' and p[3][1] == 'int':
        p[0] = p[1][0] + p[3][0] + 'ITOF\n' + 'STOREN\n'
    else:
        p[0] = p[1][0] + p[3][0] + 'STOREN\n'

def p_instruction_attribution_condition(p):
    """
    attribution : variable EQUAL condition
    """
    if p[1][1] == None:
        p[0] = f'ERR \segmentation fault\\n\\nSTOP\n'
    elif p[1][1] == 'float':
        p[0] = p[1][0] + p[3] + 'ITOF\n' + 'STOREN\n'
    else:
        p[0] = p[1][0] + p[3] + 'STOREN\n\'

def p_expression_var(p):
    """
    expression : variable
    """
    if p[1][1] == None:
        p[0] = (f'ERR \segmentation fault\\n\\nSTOP\n', None)
    else:
        p[0] = (p[1][0] + 'LOADN\n', p[1][1])

```

```

def p_expression_num(p):
    """
    expression : NUM
    """
    p[0] = (f'PUSHI {p[1]}\n','int')

def p_expression_float(p):
    """
    expression : REAL
    """
    p[0] = (f'PUSHF {p[1]}\n','float')

def p_expression_between_parenthesis(p):
    """
    expression : LPAREN expression RPAREN
    """
    p[0] = p[2]

def p_expression_plus_expression(p):
    """
    expression : expression PLUS expression
    """
    if p[1][1] == 'int' and p[3][1] == 'int':
        p[0] = (p[1][0] + p[3][0] + 'ADD\n','int')
    elif p[1][1] == 'float' and p[3][1] == 'float':
        p[0] = (p[1][0] + p[3][0] + 'FADD\n','float')
    elif p[1][1] == 'int' and p[3][1] == 'float':
        p[0] = (p[1][0] + 'ITOF\n' + p[3][0] + 'FADD\n','float')
    else:
        p[0] = (p[1][0] + p[3][0] + 'ITOF\n' + 'FADD\n','float')

def p_expression_minus_expression(p):
    """
    expression : expression MINUS expression
    """
    if p[1][1] == 'int' and p[3][1] == 'int':
        p[0] = (p[1][0] + p[3][0] + 'SUB\n','int')
    elif p[1][1] == 'float' and p[3][1] == 'float':
        p[0] = (p[1][0] + p[3][0] + 'FSUB\n','float')
    elif p[1][1] == 'int' and p[3][1] == 'float':
        p[0] = (p[1][0] + 'ITOF\n' + p[3][0] + 'FSUB\n','float')
    else:
        p[0] = (p[1][0] + p[3][0] + 'ITOF\n' + 'FSUB\n','float')

def p_expression_mul_expression(p):
    """
    expression : expression MUL expression
    """
    if p[1][1] == 'int' and p[3][1] == 'int':
        p[0] = (p[1][0] + p[3][0] + 'MUL\n','int')
    elif p[1][1] == 'float' and p[3][1] == 'float':
        p[0] = (p[1][0] + p[3][0] + 'FMUL\n','float')
    elif p[1][1] == 'int' and p[3][1] == 'float':
        p[0] = (p[1][0] + 'ITOF\n' + p[3][0] + 'FMUL\n','float')
    else:
        p[0] = (p[1][0] + p[3][0] + 'ITOF\n' + 'FMUL\n','float')

def p_expression_div_expression(p):
    """
    expression : expression DIV expression
    """

```

```

if p[1][1] == 'int' and p[3][1] == 'int':
    p[0] = (p[1][0] + p[3][0] + 'DIV\n', 'int')
elif p[1][1] == 'float' and p[3][1] == 'float':
    p[0] = (p[1][0] + p[3][0] + 'FDIV\n', 'float')
elif p[1][1] == 'int' and p[3][1] == 'float':
    p[0] = (p[1][0] + 'ITOF\n' + p[3][0] + 'FDIV\n', 'float')
else:
    p[0] = (p[1][0] + p[3][0] + 'ITOF\n' + 'FDIV\n', 'float')

def p_expression_mod_expression(p):
    """
    expression : expression MOD expression
    """
    if p[1][1] == 'int' and p[3][1] == 'int':
        p[0] = (p[1][0] + p[3][0] + 'MOD\n', 'int')
    elif p[1][1] == 'float' and p[3][1] == 'float':
        p[0] = (p[1][0] + 'FTOI\n' + p[3][0] + 'FTOI\n' + 'MOD\n', 'int')
    elif p[1][1] == 'int' and p[3][1] == 'float':
        p[0] = (p[1][0] + p[3][0] + 'FTOI\n' + 'MOD\n', 'int')
    else:
        p[0] = (p[1][0] + 'FTOI\n' + p[3][0] + 'MOD\n', 'int')

def p_instruction_while(p):
    """
    instruction : WHILE LPAREN condition RPAREN LCURLY instructions RCURLY
    """
    p[0] = f'B{parser.labels}:\n' + p[3] + 'JZ ' + f'E{parser.labels}\n' + p[6] +
    f'JUMP B{parser.labels}\n' + f'E{parser.labels}:\n'
    parser.labels += 1

def p_instruction_for(p):
    """
    instruction : FOR LPAREN atributions SEMICOLON condition SEMICOLON atributions RPAREN LCURLY instructions RCURLY
    """
    p[0] = p[3] + f'B{parser.labels}:\n' + p[5] + 'JZ ' + f'E{parser.labels}\n' + p[10] + p[7] +
    f'JUMP B{parser.labels}\n' + f'E{parser.labels}:\n'
    parser.labels += 1

def p_instruction_if(p):
    """
    instruction : IF LPAREN condition RPAREN LCURLY instructions RCURLY
    """
    p[0] = p[3] + 'JZ ' + f'E{parser.labels}\n' + p[6] + f'E{parser.labels}:\n'
    parser.labels += 1

def p_instruction_if_else(p):
    """
    instruction : IF LPAREN condition RPAREN LCURLY instructions RCURLY ELSE LCURLY instructions RCURLY
    """
    p[0] = p[3] + 'JZ ' + f'E{parser.labels}\n' + p[6] + f'JUMP F{parser.labels}\n' +
    f'E{parser.labels}:\n' + p[10] + f'F{parser.labels}:\n'
    parser.labels += 1

def p_condition_expression_eqeq_expression(p):
    """
    condition : expression EQEQ expression
    """
    if p[1][1] == 'int' and p[3][1] == 'float':
        p[0] = p[1][0] + 'ITOF\n' + p[3][0] + 'EQUAL\n'
    elif p[1][1] == 'float' and p[3][1] == 'int':
        p[0] = p[1][0] + p[3][0] + 'ITOF\n' + 'EQUAL\n'
    else:

```



```

p[0] = p[1][0] + p[3][0] + 'EQUAL\n'

def p_condition_expression_diff_expression(p):
    """
    condition : expression DIFF expression
    """
    if p[1][1] == 'int' and p[3][1] == 'float':
        p[0] = p[1][0] + 'ITOF\n' + p[3][0] + 'EQUAL\nNOT\n'
    elif p[1][1] == 'float' and p[3][1] == 'int':
        p[0] = p[1][0] + p[3][0] + 'ITOF\n' + 'EQUAL\nNOT\n'
    else:
        p[0] = p[1][0] + p[3][0] + 'EQUAL\nNOT\n'

def p_condition_expression_greater_expression(p):
    """
    condition : expression GREATER expression
    """
    if p[1][1] == 'float' and p[3][1] == 'float':
        p[0] = p[1][0] + p[3][0] + 'FSUP\n' + 'FTOI\nPUSHI 0\nSUP\n'
    elif p[1][1] == 'int' and p[3][1] == 'float':
        p[0] = p[1][0] + 'ITOF\n' + p[3][0] + 'FSUP\n' + 'FTOI\nPUSHI 0\nSUP\n'
    elif p[1][1] == 'float' and p[3][1] == 'int':
        p[0] = p[1][0] + p[3][0] + 'ITOF\n' + 'FSUP\n' + 'FTOI\nPUSHI 0\nSUP\n'
    else:
        p[0] = p[1][0] + p[3][0] + 'SUP\n'

def p_condition_expression_lesser_expression(p):
    """
    condition : expression LESSER expression
    """
    if p[1][1] == 'float' and p[3][1] == 'float':
        p[0] = p[1][0] + p[3][0] + 'FINF\n' + 'FTOI\nPUSHI 0\nSUP\n'

    elif p[1][1] == 'int' and p[3][1] == 'float':
        p[0] = p[1][0] + 'ITOF\n' + p[3][0] + 'FINF\n' + 'FTOI\nPUSHI 0\nSUP\n'

    elif p[1][1] == 'float' and p[3][1] == 'int':
        p[0] = p[1][0] + p[3][0] + 'ITOF\n' + 'FINF\n' + 'FTOI\nPUSHI 0\nSUP\n'

    else:
        p[0] = p[1][0] + p[3][0] + 'INF\n'

def p_condition_expression_greateq_expression(p):
    """
    condition : expression GREATEREQ expression
    """
    if p[1][1] == 'float' and p[3][1] == 'float':
        p[0] = p[1][0] + p[3][0] + 'FSUPEQ\n' + 'FTOI\nPUSHI 0\nSUP\n'

    elif p[1][1] == 'int' and p[3][1] == 'float':
        p[0] = p[1][0] + 'ITOF\n' + p[3][0] + 'FSUPEQ\n' + 'FTOI\nPUSHI 0\nSUP\n'

    elif p[1][1] == 'float' and p[3][1] == 'int':
        p[0] = p[1][0] + p[3][0] + 'ITOF\n' + 'FSUPEQ\n' + 'FTOI\nPUSHI 0\nSUP\n'

    else:
        p[0] = p[1][0] + p[3][0] + 'SUPEQ\n'

def p_condition_expression_lesseq_expression(p):
    """
    condition : expression LESSEQ expression
    """
    if p[1][1] == 'float' and p[3][1] == 'float':

```

```

    p[0] = p[1][0] + p[3][0] + 'FINFEQ\n' + 'FTOI\nPUSHI 0\nSUP\n'
elif p[1][1] == 'int' and p[3][1] == 'float':
    p[0] = p[1][0] + 'ITOF\n' + p[3][0] + 'FINFEQ\n' + 'FTOI\nPUSHI 0\nSUP\n'
elif p[1][1] == 'float' and p[3][1] == 'int':
    p[0] = p[1][0] + p[3][0] + 'ITOF\n' + 'FINFEQ\n' + 'FTOI\nPUSHI 0\nSUP\n'
else:
    p[0] = p[1][0] + p[3][0] + 'INFEQ\n'

def p_condition_num(p):
    """
    condition : NUM
    """
    p[0] = f'PUSHI {p[1]}\n'

def p_condition_real(p):
    """
    condition : REAL
    """
    p[0] = f'PUSHF {p[1]}\nPUSHF 0.0\nFSUP\nFTOI\nPUSHI 0\nSUP\n'

def p_condition_var(p):
    """
    condition : variable
    """
    if p[1][1] == None:
        p[0] = f'ERR \"segmentation fault\\n\\n\" \nSTOP\n'
    elif p[1][1] == 'float':

        p[0] = p[1][0] + 'LOADN\nPUSHF 0.0\nFSUP\nFTOI\nPUSHI 0\nSUP\n'
    else:
        p[0] = p[1][0] + 'LOADN\nPUSHI 0\nSUP\n'

def p_instruction_scan(p):
    """
    instruction : SCAN LPAREN variable RPAREN SEMICOLON
    """
    if p[3][1] == None:
        p[0] = f'ERR \"segmentation fault\\n\\n\" \nSTOP\n'
    elif p[3][1] == 'int':
        p[0] = p[3][0] + 'READ\nATOI\nSTOREN\n'
    else:
        p[0] = p[3][0] + 'READ\nATOF\nSTOREN\n'

def p_instruction_print_var(p):
    """
    instruction : PRINT LPAREN variable RPAREN SEMICOLON
    """
    if p[3][1] == None:
        p[0] = f'ERR \"segmentation fault\\n\\n\" \nSTOP\n'
    elif p[3][1] == 'int':
        p[0] = p[3][0] + 'LOADN\nWRITEI\n'
    else:
        p[0] = p[3][0] + 'LOADN\nWRITEF\n'

def p_instruction_println_var(p):
    """
    instruction : PRINTLN LPAREN variable RPAREN SEMICOLON
    """
    if p[3][1] == None:
        p[0] = f'ERR \"segmentation fault\\n\\n\" \nSTOP\n'
    elif p[3][1] == 'int':
        p[0] = p[3][0] + 'LOADN\nWRITEI\n'

```

```

        p[0] += 'PUSHS\\n\\n\\nWRITES\\n'
    else:
        p[0] = p[3][0] + 'LOADN\\nWRITEF\\n'
        p[0] += 'PUSHS\\n\\n\\nWRITES\\n'

def p_instruction_print_string(p):
    """
    instruction : PRINT LPAREN STRING RPAREN SEMICOLON
    """
    p[0] = f'PUSHS {p[3]}\\nWRITES\\n'

def p_instruction_println_string(p):
    """
    instruction : PRINTLN LPAREN STRING RPAREN SEMICOLON
    """
    p[0] = f'PUSHS {p[3]}\\nWRITES\\nPUSHS\\n\\n\\nWRITES\\n'

def p_error(p):
    print("Syntax error!")
    parser.success = False

parser = yacc.yacc()
parser.success = True

parser.prox_address = 0
parser.tab_id = {}
parser.labels = 0

if len(sys.argv)!=2 and len(sys.argv)!=3:
    print('Invalid number of arguments!')
    sys.exit(0)
else:
    file_input = sys.argv[1]

if not os.path.exists(file_input):
    print(f"File \"{file_input}\" not found!")
    sys.exit(0)

fp = open(file_input, 'r')
source = fp.read()
fp.close()

if len(sys.argv)==3:
    file_output = sys.argv[2]
else:
    file_output = "a.vm"

fp = open(file_output, "w")

parser.parse(source)

if parser.success:
    print("Parsing successfully completed!")

fp.close()

```