Universidade do Minho
Escola de Engenharia

Processamento de Linguagens (3º ano de MIEI)

Trabalho Prático II

Relatório - Grupo 78

Ana Luísa Lira Tomé Carneiro (A89533)

Ana Rita Abreu Peixoto (A89612)

Luís Miguel Lopes Pinto (A89506)

Maio 2021

Resumo

Este segundo trabalho prático no âmbito da unidade curricular de processamento de linguagens consistiu na elaboração de um compilador de uma linguagem imperativa com recurso a gramáticas independentes de contexto e tradutoras. Com recurso às ferramenta *yacc* e *lex* do *python*, esta gramática deve ser capaz de gerar instruções *assembly* a partir de código de uma linguagem imperativa. No presente relatório explicamos como desenvolvemos o compilador para a linguagem e as diversas produções implementadas na gramática.Além disso, também incluímos exemplos de teste para o programa desenvolvido, tendo em consideração a funcionalidade adicional selecionada.

Conteúdo

Lista de Figuras

Listings

Capítulo 1

Introdução

1.1 Enquadramento e Contextualização

Os compiladores são muito importantes na medida em que traduzem o código de uma determinada linguagem de programação de alto nível para código máquina. Este código máquina em *assembly* pode depois ser processado e interpretado pela máquina virtual de modo a concretizar as instruções realizadas na linguagem implementada.

Neste sentido, foi implementado um compilador que traduz uma linguagem imperativa ,desenhada pelo grupo, numa sequência de instruções *assembly* de modo a poder ser interpretada por uma máquina de *stack* virtual

1.2 Problema e Objetivos

Em geral, o projeto pretende aprofundar e cimentar conceitos já abordados nas aulas, tais como:

• aumentar a experiência em engenharia de linguagens e em programação generativa (gramatical), reforçando a capacidade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT);

• desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora;

• desenvolver um compilador gerando código para uma máquina de *stack* virtual. Em concreto, será usada a VM, Virtual Machine;

• utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o *Yacc*, versão PLY do *Python*, completado pelo gerador de analisadores léxicos Lex, também versão PLY do Python;

Em particular, neste trabalho prático, pretende-se gerar instruções em *assembly* que traduzem uma linguagem imperativa personalizada. As instruções suportadas por esta linguagem imperativa passam pela declaração de variáveis do tipo inteiro com as quais será possível efetuar operações aritméticas, relacionais e lógicas. Em adição, também deve ser possível a atribuição de valores de uma expressão numérica a uma variável, a leitura do *standard input* e escrita no *standard output*, efetuar instruções condicionais e, por último, instruções cíclicas. Para a especificação do ciclo a implementar, foi necessário considerar o módulo da divisão do número do grupo

por 3. Deste modo, o ciclo que nos foi atibuído foi o *while-do*. Em adição, consideramos que apenas podem ser declaradas variáveis no inicio do programa e que não é possível utiliza-las sem declaração prévia. No âmbito das funcionalidades adicionais, implementamos a definição e invocação de subprogramas sem parâmetros, que retornam um resultado do tipo inteiro.

1.3 Estrutura do documento

O presente relatório tem como objetivo ilustrar o trabalho realizado. Para isso, estruturamos o relatório em diferentes capítulos:

O primeiro capítulo é a **Introdução**. Aqui serão abordados tópicos como o enquadramento e contextualização do tema proposto, o problema que se pretende resolver e o seu objetivo e também será exposto o modo de estruturação do relatório.

De seguida, no capítulo 2 o foco será na **Análise e Especificação** do problema, onde será efetuada uma descrição informal do problema seguida da especificação dos requisitos, que permitirá abordar em detalhe as especificações dos requisitos para uma gramática GIC e GT.

No terceiro capítulo apresentamos o **Desenho da Resolução**. De forma a exemplificar a solução obtida, esta secção está subdividida em cinco temas principais: **Linguagem Imperativa** onde demonstramos um ficheiro típico utilizando a linguagem de programação criada, **Gramática com produções**, onde é exposta a gramática criada, **LEX** onde está representado os símbolos terminais da gramática, **Estrutura de dados para o YACC** onde se encontra as estruturas que sustentaram o programa e finalmente, **PLY/YACC** onde se encontra todas as ações tomadas em cada produção.

O capítulo 4 assenta na **Codificação e Testes**, que se subdivide em **Compilação e Execução**, em **Testes Realizados e Resultados na VM** e em **Decisões e Problemas de Implementação** onde estão presentes exemplos de testes, através do fornecimento de um ficheiro txt como input e a obtenção de um ficheiro txt que será usado pela VM, finalmente são discutidas as dificuldades sentidas e decisões durante a implementação.

No capítulo 5 é efetuada uma **Conclusão** e análise crítica do trabalho efetuado, realçando aspetos positivos da implementação e aspetos a melhorar.

Por fim, existe também uma última secção **Apêndice A** onde está presente o **Código do Programa**.

Capítulo 2

Análise e Especificação

2.1 Descrição Informal do Problema

Para um conjunto de instruções imperativas, independentemente da extensão do programa, pretende-se transforma-las em pseudo-código *assembly* (linguagem máquina) para posteriormente serem processadas por uma máquina virtual. A partir da leitura de cada instrução do ficheiro de *input* é possível transforma-la numa sequência de instruções *assembly* seguindo uma ordem lógica. As instruções podem ser declaração de variáveis, operações lógicas, relacionais e aritméticas entre variáveis, escrita no *stdout* e leitura do *stdin* e cíclicas. Adicionalmente também é permitida a declaração e invocação de subprogramas que retornam um resultado do tipo inteiro e não recebem argumentos.

2.2 Especificação dos Requisitos

Como forma de cumprir com o objetivo do problema apresentado é necessário analisar e especificar os dados e requisitos do projeto. Para isso, é fundamental ter em consideração as ferramentas fornecidas pelo *python*, tais como o analisador léxico ***lex*** e o analisador sintático ***yacc***. Além disso, para a implementação do programa também foi necessário ter em consideração os conceitos de **gramática independente de contexto (GIC)** e **gramática tradutora**. Por fim, tivemos também em consideração estruturas de dados do *python* como os dicionários para a implementação da solução.

Capítulo 3

Desenho da Resolução

Dada por concluída a fase de análise e especificação, chegamos agora a etapa de implementação, ou por outras palavras, desenho da resolução. Nesta etapa é importante realçar tanto o trabalho desenvolvido no analisador léxico como no analisador sintático . No primeiro, falar em particular das suas especificações (tokens, literais, ignore, etc...), e no segundo, das estruturas de dados utilizadas, da gramática que tem por base e suas produções, que consequentemente derivam em regras de tradução para Assembly da VM.

3.1 Linguagem imperativa

Para a implementação do programa foi necessário ter em conta algumas particularidades da linguagem imperativa considerada. A nossa linguagem possui uma estrutura semelhante ao C, com algumas distinções. A primeira que destacamos é o facto de que a declaração das variáveis deve ser feita no inicio do programa e deverá denotar variáveis do tipo inteiro. Além disso, no final de cada instrução não é necessário o ponto e vírgula, e não são permitidas instruções que ocupem mais de uma linha.

Alguns exemplos de programas representativos da nossa linguagem com todos os parâmetros implementados encontram-se no capitulo 4.

3.2 Gramática com produções

Para implementar o compilador foi necessário criar uma gramática tradutora de forma a conseguir interpretar uma linguagem de programação imperativa traduzindo-a em código máquina através de diversas produções. Com o analisador YACC faz um reconhecimento Bottom-Up tentou-se implementar cada produção com recursividade à esquerda, caso a produção seja recursiva. De seguida encontra-se a gramática que será utilizada pelo o compilador criado.

```
P0 : Programa -> Declaracoes Comandos Funcoes

P1 : Funcoes ->
P2 :          | Funcoes Funcao

P3 : Declaracoes ->
P4 :              | Declaracoes Declaracao

P5 : Comandos ->
P6 :           | Comandos Comando

P7 : Comando -> Atribuicao
P8 :          | Condicao
P9 :          | While
P10:          | Print

P11: Funcao -> NOME '('')''{'Programa RETURN Exp '}'

P12: Print -> PRINT '(' Factor ')'
P13:        | PRINT '( STRING ')'

P14: Declaracao -> INT id
P15:             | INT id '=' Exp
P16:             | INT id '=' READ
P17:             | INT id '=' NOME '('')'

P18: Atribuicao -> id '=' Exp
P19:             | id '=' READ
P20:             | id '=' NOME '('')'

P21: Exp -> Exp '+' Termo
P22:      | Exp '-' Termo
P23:      | Termo

P24: Termo -> Termo '*' Factor
P25:        | Termo '/' Factor
P26:        | Termo '%' Factor
P27:        | Factor

P28: Factor -> '(' Exp ')'
P29:          | num
P30:          | id

P31: Condicao -> IF '(' Comparacoes ')' Corpo
P32:           | IF '(' Comparacoes ')' Corpo ELSE Corpo

P33: While -> WHILE '(' Comparacoes ')' Corpo

P34: Corpo -> Comando
P35:         |'{' Comandos '}'
```

```
51
52 P36: Comparacoes -> Comparacao '&&' Comparacao
53 P37:              | Comparacao '||' Comparacao
54 P38:              | Comparacao
55 P39:              | '!' Comparacao
56
57 P40: Comparacao -> Factor SimbRelacional Factor
58 P41:             | '(' Comparacoes ')'
59
60 P42: SimbRelacional -> '=='
61 P43:                 | '!='
62 P44:                 | '<'
63 P45:                 | '>'
64 P46:                 | '>='
65 P47:                 | '<='
```

Listing 3.1: Produções da gramática

3.3 LEX

Para a implementação do analisador léxico foi necessário analisar e ter em consideração os símbolos terminais que a linguagem irá reconhecer. Estes símbolos são denotados por uma lista de *tokens* ou *literals* no **lex** conforme seja necessário, ou não, uma expressão regular mais complexa que não esteja implícita no próprio símbolo. Deste modo, na seguinte figura estão apresentados os símbolos literais considerados:

```
1 literals = ['(', ')', '+', '-', '*', '/', '=', '<', '>', '{', '}', '!','[',']', '%', '"']
```

Listing 3.2: Símbolos literais

Analogamente, seguem os símbolos não literais (*tokens*):

```
1 tokens = ['num','id','INT','IF','EQ','DIF','AND', 'OR','SUPEQ','INFEQ', 'ELSE', 'WHILE','
      READ','PRINT','NOME', 'RETURN', 'STRING']
```

Listing 3.3: Símbolos não literais

Deste modo, frases da nossa linguagem imperativa vão ser constituídas por símbolos **literais** como parêntesis (curvos ou retos), chavetas, operadores aritméticos como adição, subtração, multiplicação e divisão, operadores de relações lógicas para denotar o maior ou menor, operador de cálculo de resto de uma divisão e, por fim, o operador para denotar a negação de uma operação lógica.

Adicionalmente, os símbolos não terminais definidos como *tokens* são aqueles que pretendem denotar um número, identificador de variável, identificador de inteiro, sequência de caracteres que identificam as operações de *print*, *read*, *while*, *if*, *return*, *else*, nome de função e *string*. De seguida estão apresentadas as definições no *lex* de cada um destes *tokens* com a respetiva expressão regular:

```
1 t_EQ = r'=='
2 t_DIF = r'!='
3 t_AND = r'&&'
4 t_OR = r'\|\|'
```

```
t_SUPEQ = r'>='
t_INFEQ = r'<='


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

def t_RETURN(t):
    r'return'
    return t

def t_READ(t):
    r'read\(\)'
    return t

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

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

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

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

def t_NOME(t):
    r'[A-Za-z]+\(\)'
    return t

def t_STRING(t):
    r'\"[A-Za-z]+(:)?\"'
    return t

def t_id(t):
    r'[a-z]'
    return t

def t_num(t):
    r'(\-)?\d+'
    return t
```

Listing 3.4: Definição símbolos não literais (*tokens*)

3.4 Estruturas de Dados para o YACC

Para a implementação do programa foi necessário considerar algumas estruturas de dados, entre as quais 2 dicionários e 2 variáveis do tipo inteiro.

As variáveis do tipo inteiro tem como objetivo associar um número às instruções cíclicas ou condicionais, de modo a permitir o *jump* para a *label* correta. Deste modo, por cada instrução encontrada, a respetiva variável é incrementada, permitindo múltiplas instruções cíclicas ou condicionais sem conflitos.

No caso dos dicionários, o *registers* tem como propósito representar as variáveis declaradas no programa incluindo as variáveis de retorno de cada função, relacionando o seu id com a sua posição na *stack*. Por cada variável declarada é adicionado um novo registo ao dicionário correspondente a essa mesma variável e à sua localização na pilha. Por último, foi necessário implementar o dicionário *funcao* na funcionalidade adicional, com o propósito de registar qual o identificador de cada função, para novamente gerar corretamente as *labels*.

3.5 PLY / YACC

Com recurso ao analisador sintático, YACC, foi realizado o reconhecimento da gramática implementada. Assim, para cada produção reconhecida era necessário traduzi-la numa ação que dependerá de produção para produção. Esta acção, de grosso modo, será armazenar no *parser* do analisador sintático quais as operações em modo string que a máquina virtual terá realizar de forma a resolver cada produção. Após o fim do reconhecimento o conjunto de ações tomadas serão armazenado no ficheiro *output.txt* que posteriormente será interpretado pela máquina virtual devolvendo o resultado do programa criado pelo o utilizador. De seguida será abordado cada símbolo não terminal implementado com mais detalhe.

3.5.1 Programa

O primeiro símbolo não terminal é o símbolo "Programa", pelo que denota o axioma da gramática. Deste modo, o programa é constituído por uma sequência de Declarações, Comandos e Funções, tal como representado pela seguinte regra de produção:

```
1 def p_Programa(p):
2     "Programa : Declaracoes Comandos Funcoes"
3     p[0] = p[1] + p[2] + p[3]
```

Listing 3.5: Axioma da gramática

Além disso, é possível observar a ação semântica efetuada de cada vez que é reconhecido este símbolo não terminal.

3.5.2 Declarações

Cada programa criado com a linguagem implementada inicia-se sempre com as declarações das variáveis a serem usadas. Este símbolo não terminal pretende denotar a existência de uma lista de declarações, explicitando um caso de paragem e o caso da recursividade à esquerda, com recurso ao símbolo não terminal "Declaracao", tal como explicitado de seguida.

```
def p_Declaracoes_Empty(p):
    "Declaracoes :  "
    p[0] = ""

def p_Declaracoes_Declaracao(p):
    "Declaracoes : Declaracoes Declaracao"
    p[0] = p[1] + p[2]
```

Listing 3.6: Lista de Declarações

3.5.3 Comandos

Este símbolo não terminal pretende denotar a existência de uma lista de comandos, explicitando um caso de paragem e o caso da recursividade à esquerda, com recurso ao símbolo não terminal "Comando", tal como explicitado de seguida:

```
def p_Comandos_empty(p):
    "Comandos : "
    p[0] = ""

def p_Comandos_Comando(p):
    "Comandos : Comandos Comando"
    p[0] = p[1] + p[2]
```

Listing 3.7: Lista de comandos

É de notar a ação semântica associada a este símbolo terminal, que é constituída pelas ações de cada símbolo não terminal reconhecido.

3.5.4 Comando

O símbolo "Comando"subdivide-se em 4 outros símbolos. Desta forma, este símbolo pode denotar uma atribuição, uma condição, uma instrução cíclica ou de escrita no *stdout*. As regras de produção para este símbolo e as respetivas ações semânticas estão apresentadas de seguida:

```
def p_Comando_Atribuicao(p):
    "Comando : Atribuicao"
    p[0] = p[1]

def p_Comando_Condicao(p):
    "Comando : Condicao"
    p[0] = p[1]

```

```
def p_Comando_While(p):
    "Comando : While"
    p[0] = p[1]

def p_Comando_Print(p):
   "Comando : Print"
   p[0] = p[1]
```

Listing 3.8: Regras de produção Comando

3.5.5 Funções

Cada programa criado poderá invocar e definir subprogramas que serão representados como funções. O símbolo não terminal "Funcoes"representa a existência de um conjunto de funções que poderam ser definidas no programa, explicitando um caso de paragem e o caso da recursividade à esquerda, com recurso ao símbolo não terminal "Funcao", tal como explicitado de seguida. De forma a limitar a secção de cada função criou-se uma *label endFunction* que indica o fim da secção de funcoes. Assim, quando o programa tiver acabado este identifica os *jumps* para as *label* de fim de função e por isso o código de cada função não é percorrido.

```
def p_Funcoes_Empty(p):
    "Funcoes : "
    p[0] = ""

def p_Funcoes_Funcoes(p):
   "Funcoes : Funcoes Funcao"
   p[0] = p[1] + "jump endFunction\n" + p[2] + "endFunction:\n"
```

Listing 3.9: Lista de Funções

3.5.6 Funcao

Como forma a invocar e definir cada função foi criado a seguinte acão associada à produção representativa de uma função.

```
def p_Funcao(p):
    "Funcao : NOME '{' Programa RETURN Exp '}' "

    n = p.parser.registers.get(p[1])
    prog = p.parser.funcao.get(p[1])


    p[0] = "subProgram" + str(prog) +":\n" + p[3] + p[5] + "STOREG " + str(n) + "\n" + "jump
     final" + str(prog) + "\n" + "endProgram" + str(prog) +":\n"
```

Listing 3.10: Acão associada a cada função

Cada função é constituída por um nome, um programa retornando no final uma expressão. Numa primeira instância vamos buscar ao dicionário registos qual o índice onde se encontra a variável de retorno. À medida que cada função é invocada no programa vamos recorrer a *jumps* e *labels* como forma de delimitar o inicio

de cada subprograma. É por isso necessário determinar qual a *label* de cada função para posteriormente ser usada nos *jumps*. As *labels* serão obtidas com recurso ao dicionário *stackFuncao* que está a ser preenchido à medida que invocamos as funções. Por exemplo se a função "sum"é invocada então a sua *label* de inicio de conteúdo será "subProgram0"caso esta tenha sido a primeira função a ser invocada, "subProgram1"caso tenha sido a segunda e assim sucessivamente. De seguida vamos colocar em p[0], e posteriormente no ficheiro, quais as operações que a máquina virtual deverá realizar. Sequencialmente as ações seriam delimitar a função através das *labels* calculadas, realizar as operações de cada função, isto é, o programa e calcular o valor de retorno que consequentemente será armazenado na variável de retorno da função. Finalmente, é necessário que a máquina virtual retorne novamente ao local do código onde estava aquando da invocação. Isso é feito recorrendo novamente a *labels* e *jumps* que fazem com que o apontador da *stack* volte ao inicio da invocação.

3.5.7 Print

Como forma de devolver resultados pelo o output ao utilizador foi desenvolvida uma função print() que recebe como argumento um inteiro, quer este seja o resultado de uma expressão que seja um único elemento, ou uma *string* escrevendo no *output* o argumento recebido. As ações associadas a cada argumento estão representadas na imagem seguinte.

```
1
2 def p_Print(p):
3     "Print : PRINT '(' Exp ')'"
4     #string = '"\n"'
5     p[0] = p[3] + "WRITEI\n"
6
7 def p_Print_string(p):
8     "Print : PRINT '(' STRING ')'"
9     #string = '"\n"'
10    p[0] = "PUSHS " + p[3] + "\nWRITES\n"
```

Listing 3.11: Acão associada ao print

Como forma de traduzir o *print* em linguagem máquina foi utilizado a operação WRITES/WRITEI disponível na máquina virtual, assim como o PUSHS que coloca na *heap* uma *string*.

3.5.8 Declaracao

Uma declaração possui 4 vertentes, ou seja, é possível simplesmente declarar a variável a ser usada atribuir a uma variável o valor de uma expressão, o valor de leitura do *stdin* ou até o valor de retorno de uma função, através da sua invocação.

No caso de uma simples declaração de variável, tal como está referido no enunciado, assumimos que o valor inicial desta é 0. Para isso vamos armazenar na *stack* de registos o registo inicial desta variável implementando uma nova operação máquina, PUSHI 0.

No caso da atribuição da expressão a uma variável, são primeiramente efetuadas as ações semânticas relativas à expressão. Desta forma, o valor da variável na *stack* será correspondente ao resultado da expressão.

A segunda operação de atribuição corresponde ao caso em que se pretende declarar uma variável com o valor obtido por input. Em linguagem máquina são geradas uma sequência de instruções que começam por um READ, seguido de um ATOI (para converter o valor para inteiro, dado que apenas existem declarações de valores inteiros no programa) e por último o STOREG do resultado na localização da variável pretendida.

Por último, foi necessário declarar uma variável com o valor de retorno de uma função. Para esta declaração é necessário ter em consideração o identificador daquela função no dicionário de funções de forma a determinar as *labels* corretas e armazenar espaço na *stack* para o resultado de retorno da função através do dicionário de registos. As instruções *assembly* utilizadas correspondem à alocação de espaço tanto da variável declarada como para o retorno da função. De seguida estão implementadas as instruções correspondentes ao código da função respetiva incluindo os *jumps* e *labels* necessários ao bom funcionamento do programa. Assim, uma vez que seja invocada uma função é realizado um *jump* que coloca o apontador no local da *stack* onde começa a função invocada. Após a realização da função é necessário a criação de *label* que permita à maquina virtual voltar ao momento da invocação e continuar com o programa. Para isso utiliza-se a *label* final que é atingida mal a acabe a função invocada. É por fim necessário armazenar o resultado da função na variável declarada.

De seguida está o código em *python* relativo às 4 declaracoes explicitadas acima e as suas produções:

```
def p_Declaracao_id(p):
    "Declaracao : INT id"
    n = len(p.parser.registers)
    p.parser.registers.update({p[2]: n})
    p[0] = "PUSHI 0" + "\n"

def p_Declaracao_atrib(p):
    "Declaracao : INT id '=' Exp"
    n = len(p.parser.registers)
    p.parser.registers.update({p[2]: n})
    p[0] = p[4] + "\n"

def p_Declaracao_read(p):
    "Declaracao : INT id '=' READ"
    n = len(p.parser.registers)
    p.parser.registers.update({p[2]: n})
    p[0] = "PUSHI 0\n" + "READ\n" + "ATOI\n" + "STOREG " + str(n) + "\n"


def p_Declaracao_funcao(p):
    "Declaracao : INT id '=' NOME"

    n = len(p.parser.registers)
    p.parser.registers.update({p[2]:n})

    prog = len(p.parser.funcao)
    p.parser.funcao.update({p[4]: prog})

    tam = len(p.parser.registers)
    p.parser.registers.update({p[4]:tam})

```

```
32      p[0] = "PUSHI 0\n" + "PUSHI 0\n" + "jump subProgram" + str(prog) +"\n" + "final" + str(
        prog) +":\n" +"PUSHG " + str(tam) +"\n" + "STOREG " + str(n) + "\n"
```

Listing 3.12: Acão associada ao declaracao

3.5.9 Atribuição

Tal como foi anteriormente referido, um "Comando"pode denotar uma "Atribuicao". Uma atribuição possui várias vertentes, ou seja, é possível atribuir a uma variável o valor de uma expressão, o valor de leitura do *stdin* ou até o valor de retorno de uma função, através da sua invocação.

No caso da atribuição da expressão a uma variável, são primeiramente efetuadas as ações semânticas relativas à expressão e de seguida uma instrução STOREG para a localização na *stack* correspondente a essa mesma variável. Desta forma, o valor da variável na *stack* será atualizado para o novo valor (resultado da expressão). Nesta atribuição também são tratados casos em que poderia ocorrer erro, ou seja, não é permitido que o utilizador efetue uma atribuição caso a variável não esteja declarada.

A segunda operação de atribuição corresponde ao caso em que se pretende atribuir a uma variável o valor obtido por uma operação de leitura do *stdin*. O procedimento para o tratamento de erros é análogo ao anteriormente referido. Em linguagem máquina são geradas uma sequência de instruções que começam por um READ, seguido de um ATOI (para converter o valor para inteiro, dado que apenas existem declarações de valores inteiros no programa) e por último o STOREG do resultado na localização da variável pretendida.

Por último, foi necessário efetuar uma operação de atribuição no caso de invocações de funções, de modo a atribuir o seu valor de retorno a uma variável do programa. O tratamento de erros para o caso em que a variável não existe é análogo aos dois casos anteriores. Para a atribuição propriamente dita, no caso em que é possível efetuá-la, é necessário ter em consideração o número identificador daquela função e a localização da variável na *stack*, obtidos com recurso aos dicionários. As instruções *assembly* utilizadas, para além de corresponderem à alocação de espaço na *stack* para o retorno da função, denotam o salto para o programa quando se depara com a invocação de uma função. A instrução "jump subProgram"denota o salto para o conteúdo do programa. Esta instrução é seguida da *label* "final"para o programa poder retroceder ao ponto em que estava quando foi invocado. De seguida, são efetuadas operações de PUSHG e STOREG com o propósito de ir buscar o valor de retorno da função e de o armazenar na variável pretendida, respetivamente.

De seguida está o código em *python* relativo às 3 atribuições explicitadas acima e as suas produções:

```
1  def p_Atribuicao(p):
2      "Atribuicao : id '=' Exp "
3      n = p.parser.registers.get(p[1])
4      if(n==None):
5          p[0] = "ERRO: ID sem definicao\n"
6      else:
7          p[0] = p[3] + "STOREG " + str(n) + "\n"
8
9  def p_Atribuicao_read(p):
10     "Atribuicao : id '=' READ "
11     n = p.parser.registers.get(p[1])
12     if(n==None):
```

```
        p[0] = "ERRO: ID sem defenicao\n"
        p.parser.s
    else:
        p[0] = "READ\n" + "ATOI\n" + "STOREG " + str(n) + "\n"

def p_Atribuicao_funcao(p):
    "Atribuicao : id '=' NOME"

    n = p.parser.registers.get(p[1])

    if(n==None):
        p[0] = "ERRO: ID sem defenicao\n"
    else:
        prog = len(p.parser.funcao)
        p.parser.funcao.update({p[3]:prog})

        tam = len(p.parser.registers)
        p.parser.registers.update({p[3]:tam})

        p[0] = "PUSHI 0\n" + "jump subProgram" + str(prog) +"\n" + "final" + str(prog) +":\n
    " + "PUSHG " + str(tam) + "\n" + "STOREG " + str(n) + "\n"
```

Listing 3.13: Regras de produção Atribuicao

3.5.10 Expressões

O símbolo não terminal "Exp"denota as expressões que podem ser efetuadas no programa, tais como a adição e subtração. Deste modo, uma expressão pode ser um "Termo"ou uma adição/subtração de uma "Exp"com um "Termo". Além disso, também pode denotar um elemento "Termo". Este símbolo não terminal não inclui as operações de multiplicação e divisão de forma a respeitar a prioridade que estas possuem nas operações de cariz matemático. De seguida estão as regras de produção relativas a este símbolo não terminal:

```
def p_Exp_termo(p):
    "Exp : Termo"
    p[0] = p[1]

def p_Exp_add(p):
    "Exp : Exp '+' Termo"
    p[0] = p[1] + p[3] + "ADD\n"

def p_Exp_sub(p):
    "Exp : Exp '-' Termo"
    p[0] = p[1] + p[3] + "SUB\n"
```

Listing 3.14: Regras de produção Exp

3.5.11 Termo

Neste símbolo não terminal estão presentes as operações de divisão, multiplicação e de cálculo do resto de uma divisão. Deste modo, este símbolo não terminal pode ser caracterizado por cada uma destas operações ou por um "Factor". Para cada uma das operações foram implementadas ações semânticas distintas.

No caso do resto da divisão, a frase reconhecida deverá ser do tipo "Termo % Factor". A título exemplificativo, uma possível frase seria 10 % 2. A ação semântica associada tem em consideração a ação do "Termo"e do "Factor"reconhecidos com o acréscimo da instrução máquina MOD para ser processada pela máquina virtual.

Para a multiplicação, o procedimento é semelhante sendo que neste caso o operador é o e denota a multiplicação entre um "Termo"e um "Factor". A ação semântica tem em consideração o resultado do reconhecimento dos símbolos não terminais constituintes seguida da operação MUL em *assembly*.

Na divisão foi implementado o mesmo raciocínio. O operador neste caso é o e pretende denotar a divisão entre um "Termo"e um "Factor". A ação semântica tem em consideração as ações de cada um destes símbolos seguida da instrução DIV em linguagem máquina. É de notar que se está a ter em consideração o tratamento de erros para o caso da divisão por zero.

Por último, consideramos também que um "Termo"pode ser também um "Factor"(abordado com mais detalhe na respetiva secção).

De seguida estão as regras de produções e respetivas ações semânticas relativa são símbolo não terminal "Termo":

```
def p_Termo_mod(p):
    "Termo : Termo '%' Factor"
    p[0] = p[1] + p[3] + "MOD\n"

def p_Termo_mul(p):
    "Termo : Termo '*' Factor"
    p[0] = p[1] + p[3] + "MUL\n"

def p_Termo_div(p):
    "Termo : Termo '/' Factor"
    if (p[3]!=0):
        p[0] = p[1] + p[3] + "DIV\n"
    else:
        print("Erro: Divisao por 0, a continuar com 0")
        p[0] = "ERRO"

def p_Termo_factor(p):
    "Termo : Factor"
    p[0] = p[1]
```

Listing 3.15: Regras de produção Termo

3.5.12 Factor

O próximo símbolo não terminal é o símbolo "Factor". Este símbolo pretende reconhecer uma expressão entre parêntesis curvos ou então os símbolos terminais "num"e "id".

Ao reconhecer uma expressão entre parêntesis curvos procede às ações semânticas relativas a essa mesma expressão.

No caso do símbolo terminal "num"reconhecido, a ação semântica a tomar é efetuar um PUSHI para a *stack* desse mesmo número lido.

Para o símbolo terminal "id"o procedimento é análogo, com a diferença de que fazemos um PUSHG para armazenar a posição em que o valor da variável vai ser armazenado na *stack*. Para obter esta posição recorre-se ao dicionário "registers".

De seguida estão apresentadas as regras de produção e ações semânticas relativas a este símbolo não terminal:

```
def p_Factor_group(p):
    "Factor : '(' Exp ')'"
    p[0] = p[2]

def p_Factor_num(p):
    "Factor : num"
    p[0] = "PUSHI " + p[1] + "\n"

def p_Factor_id(p):
    "Factor : id"
    n = p.parser.registers.get(p[1])
    p[0] = "PUSHG " + str(n) + "\n"
```

Listing 3.16: Regras de produção Factor

3.5.13 Condição

Este símbolo não terminal pretende denotar as instruções condicionais suportadas pelo programa. Deste modo, existem dois casos a considerar: o caso *if then* e o caso *if then else*.

Para o primeiro caso, a frase reconhecida dá origem a código máquina que começa com a *label* "ifstmnt"associada ao identificador daquela condição, obtido com recurso à variável "statement"do *yacc*. Esta *label* é seguida das instruções correspondentes às "Comparacoes"e caso não se verifique a sua veracidade é efetuado um salto (jz endif) para avançar para o final da declaração da condição.

No caso do *if then else* o procedimento foi análogo mas neste caso temos que tratar também o caso do *else*. Para isso, foram implementadas diferentes *labels* para fazer os saltos corretamente, para o caso em que as condições são verdadeiras ou falsas. Para o tratamento das *labels* foi necessário ter em consideração o número que identifica aquela instrução condicional. É de notar que por cada instrução condicional reconhecida a variável "statement"é incrementada em 1 unidade. De seguida está apresentado o código em *python* relativo ao símbolo terminal "Condicao"e respetivas ações semânticas:

```
def p_Condicao_If(p):
    "Condicao : IF '(' Comparacoes ')' Corpo"
    n = p.parser.statement
    p.parser.statement = n + 1
    p[0] = "ifstmnt" + str(n) + ":\n" + p[3] + "jz endif" + str(n) +"\n" + p[5] + "endif" +
    str(n) + ":\n"
```

```
6
7  def p_Condicao_IfElse(p):
8      "Condicao : IF '(' Comparacoes ')' Corpo ELSE Corpo"
9      n = p.parser.statement
10     p.parser.statement = n + 1
11     p[0] = "ifstmnt" + str(n) + ":\n" + p[3] + "jz endif" + str(n) +"\n" + p[5] + "jump end"
        + str(n) + "\n" + "endif" + str(n) + ":\n" + p[7] + "end" + str(n) + ":\n"
```

Listing 3.17: Regras de produção Condicao

3.5.14 While

Para o caso do *while* foi implementada uma produção que reconhece instruções deste tipo na linguagem imperativa. Começa-se por verificar qual o identificador do ciclo com recurso à variável "loop"que efetua a contagem de todos os ciclos do programa e atribui um número a cada um deles, para efetuar as *labels* corretamente. Além disso, existe uma ação semântica associada que apresenta as instruções *assembly* geradas quando é reconhecida uma instrução cíclica. Primeiramente coloca-se a *label* "repeat"associada ao respetivo identificador do ciclo, seguidas das condições do ciclo "Comparacoes"e da instrução "jz endRepeat"para controlo do ciclo. Caso as condições sejam verdadeiras, é efetuado o corpo do ciclo. Caso contrário, o corpo do ciclo é avançado. Esta dinâmica é assegurada pelas diferentes *labels* e instruções de salto geradas. A implementação traduziu-se em *python* da seguinte forma:

```
1  def p_While(p):
2      "While : WHILE '(' Comparacoes ')' Corpo"
3      n = p.parser.loop
4      p.parser.loop = n + 1
5      p[0] = "repeat" + str(n) + ":\n" + p[3] + "jz endRepeat" + str(n) +"\n" + p[5] + "jump
       repeat" + str(n) + "\n" + "endRepeat" + str(n) + ":\n"
```

Listing 3.18: Regras de produção While

3.5.15 Corpo

Este símbolo não terminal tem como objetivo denotar as instruções que podem constituir o corpo de um ciclo ou de uma instrução condicional na linguagem imperativa considerada.

Deste modo, um "Corpo"pode ser apenas um "Comando"ou uma lista de comandos denotada pelo símbolo terminal "Comandos". No último caso é necessário possuir chavetas em torno do símbolo não terminal para respeitar a sintaxe da linguagem. Esta implementação é visível no código em *python* apresentado de seguida:

```
1  def p_Corpo_Comando(p):
2      "Corpo : Comando"
3      p[0] = p[1]
4
5  def p_Corpo_Comandos(p):
6      "Corpo : '{' Comandos '}'"
7      p[0] = p[2]
```

Listing 3.19: Regras de produção Corpo

3.5.16 Comparacoes

Para as condições de controlo de ciclo e para as instruções condicionais foi necessário implementar um símbolo não terminal "Comparacoes"com o propósito de albergar todos os casos possíveis suportados pelo nosso programa. Deste modo, foram implementadas operações de conjunção, disjunção, distinção ou simples.

Para a conjunção considerou-se o símbolo terminal AND e a operação semântica associada teve em consideração as operações semânticas de cada símbolo não terminal constituinte seguido da operação de multiplicação representada pela instrução MUL em *assembly*.

No caso da disjunção, a frase a reconhecer denota uma disjunção entre duas comparações com o auxílio do símbolo terminal OR. As ações semânticas associadas tem em consideração as ações de cada símbolo não terminal com o auxílio das instruções ADD, MUL e SUB do *assembly* com o propósito de apurar a veracidade daquela comparação.

Em adição, foi também considerado o caso de negação de uma comparação em que se recorre ao símbolo terminal '!' e à instrução em *assembly* NOT.

Por último, consideramos que o símbolo não terminal "Comparacoes"pode denotar também uma única "Comparacao".

De seguida está apresentado o código que deu sustento ao referido anteriormente:

```
def p_Comparacoes_Conj(p):
    "Comparacoes : Comparacao AND Comparacao"
    p[0] = p[1] + p[3] + "MUL\n"

def p_Comparacoes_Disj(p):
    "Comparacoes : Comparacao OR Comparacao"
    p[0] = p[1] + p[3] + "ADD\n" + p[1] + p[3] + "MUL\n" + "SUB\n"

def p_Comparacoes_Comparacao(p):
    "Comparacoes : Comparacao"
    p[0] = p[1]

def p_Comparacoes_NotComparacao(p):
    "Comparacoes : '!' Comparacao"
    p[0] = p[2] + "NOT\n"
```

Listing 3.20: Regras de produção Comparacoes

3.5.17 Comparacao

Uma comparação simples ou conjunto de comparações envolvidas por parêntesis é denotado pelo não terminal "Comparacao".

Para o caso da comparação simples, a frase reconhecida deverá obedecer à seguinte sequência "Factor SimbRelacional Factor"e a ação semântica associada corresponde às ações semânticas de cada um destes símbolos não terminais.

O caso da comparação composta é aquele em que podemos ter em conta várias comparações envolvidas por parêntesis, recorrendo ao não terminal "Comparacoes"que irá denotar as suas ações semânticas.

Podemos observar de seguida o código em *python* relativo ao não terminal "Comparacao"e as respetivas ações semânticas:

```
def p_Comparacao_Simples(p):
    "Comparacao : Factor SimbRelacional Factor"
    p[0] = p[1] + p[3] + p[2]

def p_Comparacao_Composta(p):
    "Comparacao : '(' Comparacoes ')'"
    p[0] = p[2]
```

Listing 3.21: Regras de produção Comparacao

3.5.18 SimbRelacional

Por último, o símbolo não terminal "SimbRelacional"pretende agrupar os diferentes símbolos terminais relativos a uma operação lógica, como por exemplo os símbolos de igualdade ou maior.

Para cada símbolo terminal reconhecido é efetuada uma ação semântica com o propósito de gerar o código *assembly* correspondente à frase reconhecida.

As regras de produção e ações semânticas deste símbolo não terminal estão apresentadas de seguida em *python*:

```
def p_SimbRelacional_EQ(p):
    "SimbRelacional : EQ"
    p[0] = "EQUAL\n"

def p_SimbRelacional_DIF(p):
    "SimbRelacional : DIF"
    p[0] = "EQUAL\nNOT\n"

def p_SimbRelacional_INF(p):
    "SimbRelacional : '<'"
    p[0] = "INF\n"

def p_SimbRelacional_INFEQ(p):
    "SimbRelacional : INFEQ"
    p[0] = "INFEQ\n"

def p_SimbRelacional_SUPEQ(p):
    "SimbRelacional : SUPEQ"
    p[0] = "SUPEQ\n"

def p_SimbRelacional_SUP(p):
    "SimbRelacional : '>'"
    p[0] = "SUP\n"
```

Listing 3.22: Regras de produção SimbRelacional

Capítulo 4

Codificação e Testes

4.1 Compilação e Execução

O programa implementado está capacitado para efetuar a leitura linha a linha a partir de código inserido no *stdin* ou então é possível passar-lhe um ficheiro para a compilação, com recurso à linha de comandos, da seguinte forma:

```
cat input.txt | python3.9 compilador_yacc.py
```

4.2 Testes Realizados e Resultados na VM

De forma a por à prova o programa implementado foram efetuados alguns testes. Para decidiu-se implementar alguns programas escritos na linguagem imperativa criada e para verificar o resultado em *assembly* obtido a partir da compilação do programa utilizando a gramática implementada. Após essa compilação, foi obtido um ficheiro em código máquina que foi testado na máquina virtual para garantir um resultado correto do programa.

4.2.1 Lados de um quadrado

Um dos testes realizados passou por construir um programa que a partir de 4 valores lidos por input determine se podem ser comprimentos de lados de um quadrado. Implementou-se, assim, o seguinte ficheiro com o respetivo código.

```
1 int a = read()
2 int b = read()
3 int c = read()
4 int d = read()
5
6 if ( a==b && (b==c && c==d)) print("true") else print ( "false" )
```

Listing 4.1: Teste1

Após a compilação, este ficheiro devolveu o seguinte resultado output com o código máquina. compilador

```
PUSHI 0
READ
ATOI
STOREG 0
PUSHI 0
READ
ATOI
STOREG 1
PUSHI 0
READ
ATOI
STOREG 2
PUSHI 0
READ
ATOI
STOREG 3
ifstmnt0:
PUSHG 0
PUSHG 1
EQUAL
PUSHG 1
PUSHG 2
EQUAL
PUSHG 2
PUSHG 3
EQUAL
MUL
MUL
jz endif0
PUSHS "true"
WRITES
jump end0
endif0:
PUSHS "false"
WRITES
end0:
```

Listing 4.2: Output do teste 1

Utilizando como input os valores 2,2,2,2 obteve-se o resultado *true*, ou seja, os lados formam um comprimento.

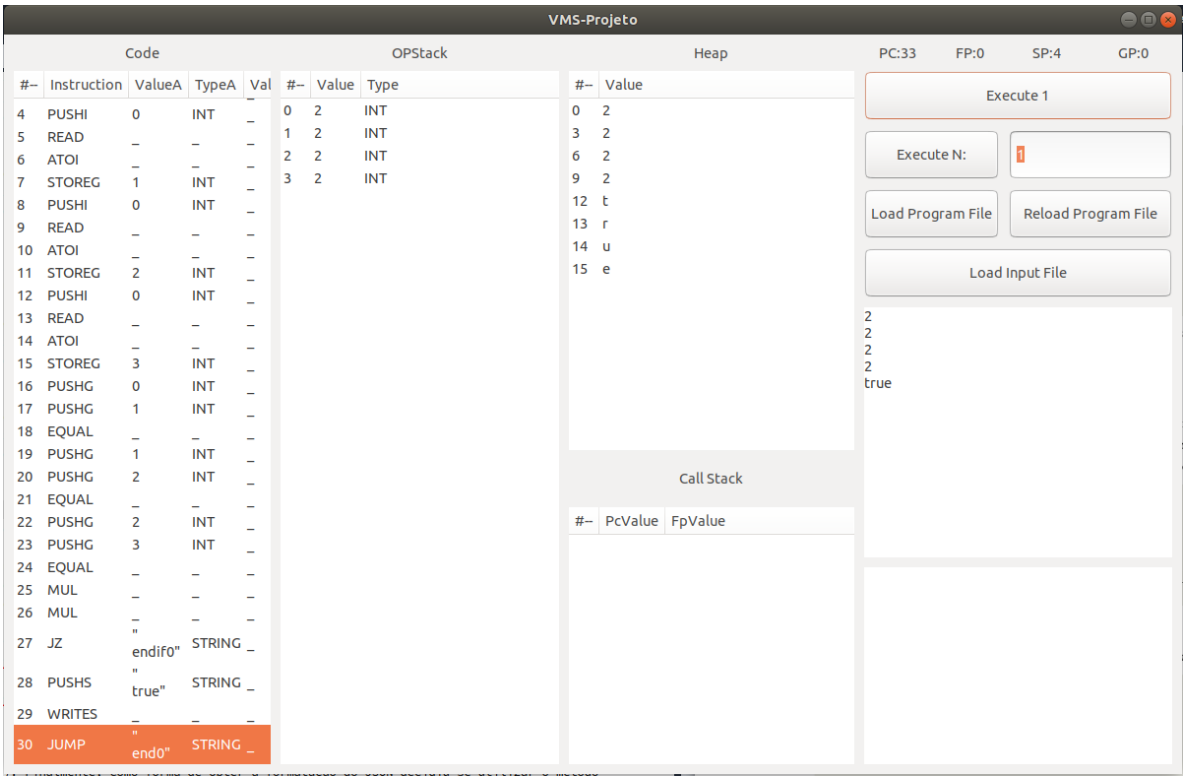

Figura 4.1: Resultado do teste 1 na VM

Utilizando como input os valores 2,3,2,3 obteve-se o resultado *false*, ou seja, os lados não formam um comprimento.

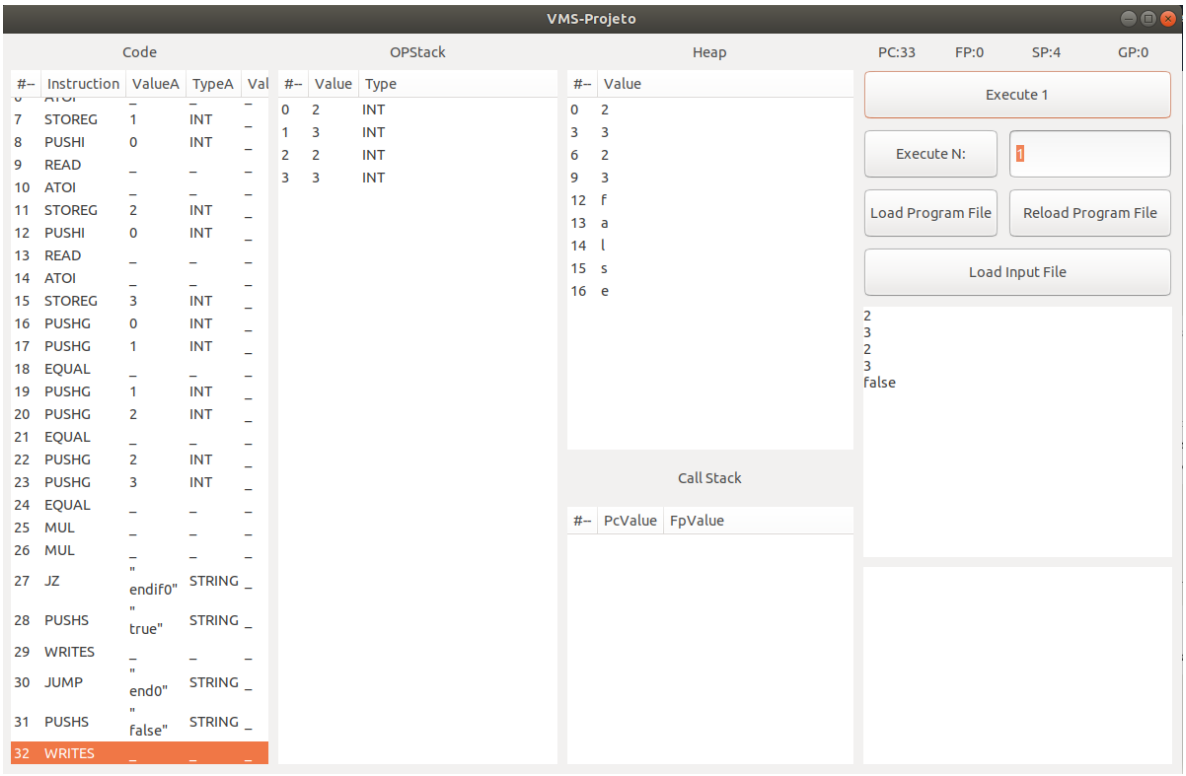

Figura 4.2: Segundo resultado do teste 1 na VM

4.2.2 Menor valor de um conjunto de valores

Para o segundo dos testes realizados implementou-se um programa que a partir de um conjunto N de valores lidos por input determine o menor desse conjunto. Implementou-se, assim, o seguinte ficheiro com o respetivo código.

```
int n = read()
int a = read()
int b
n = n- 1

while(n != 0) { b = read() if(b<a){a = b} n=n- 1}

print("Menor:")
print(a)
```

Listing 4.3: Teste2

Após a compilação, este ficheiro devolveu o seguinte resultado output com o código máquina. compilador

```
PUSHI 0
READ
ATOI
STOREG 0
PUSHI 0
READ
ATOI
STOREG 1
PUSHI 0
PUSHG 0
PUSHI 1
SUB
STOREG 0
repeat0:
PUSHG 0
PUSHI 0
EQUAL
NOT
jz endRepeat0
READ
ATOI
STOREG 2
ifstmnt0:
PUSHG 2
PUSHG 1
INF
jz endif0
PUSHG 2
STOREG 1
endif0:
PUSHG 0
PUSHI 1
SUB
STOREG 0
jump repeat0
endRepeat0:
PUSHS "Menor:"
WRITES
PUSHG 1
WRITEI
```

Listing 4.4: Output do teste 2

Utilizando como input os seguintes 5 valores, 2,3,4,2,1 determinou-se que o menor valor é o 1 o que está correto.

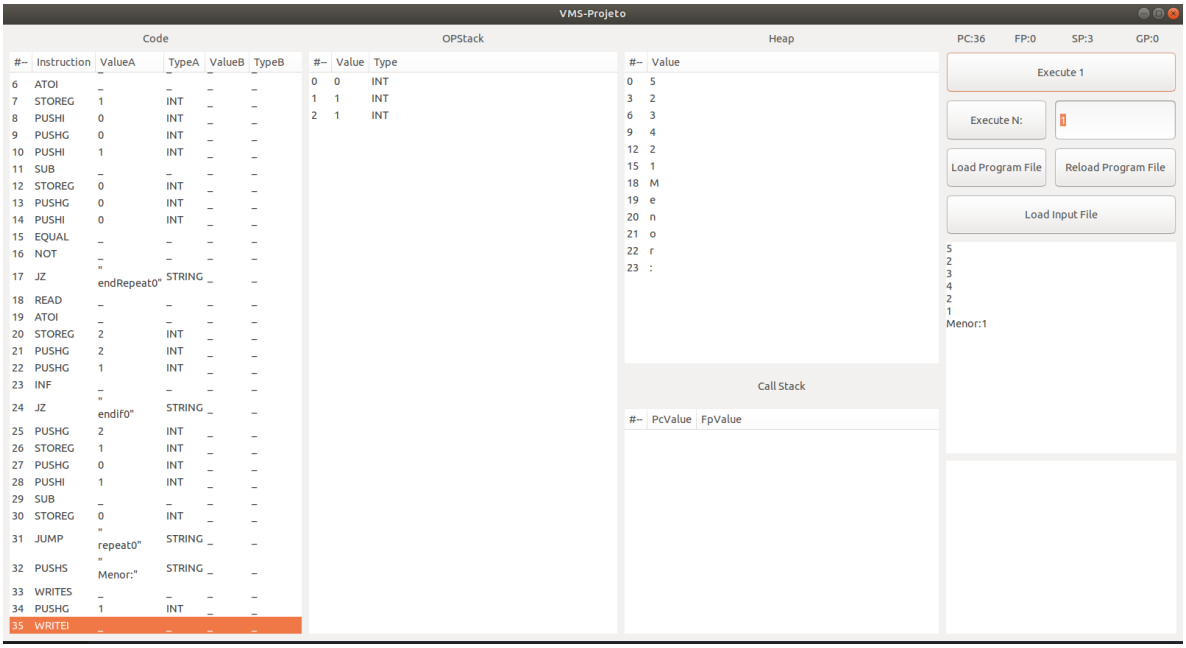

Figura 4.3: Resultado do teste 2 na VM

4.2.3 Produtório de um conjunto de números

No terceiro teste realizado implementou-se um programa que a partir de um conjunto 4 de valores lidos por input determine o produtório desse conjunto. Implementou-se, assim, o seguinte ficheiro com o respetivo código.

```
int a = 4
int r = 1
int b


while (a != 0) { b=read() r = r*b a = a- 1}

print("Produtorio:")
print(r)
```

Listing 4.5: Teste3

Após a compilação, este ficheiro devolveu o seguinte resultado output com o código máquina. compilador

```
PUSHI 4

PUSHI 1

PUSHI 0
repeat0:
```

```
PUSHG 0
PUSHI 0
EQUAL
NOT
jz endRepeat0
READ
ATOI
STOREG 2
PUSHG 1
PUSHG 2
MUL
STOREG 1
PUSHG 0
PUSHI 1
SUB
STOREG 0
jump repeat0
endRepeat0:
PUSHS "Produtorio:"
WRITES
PUSHG 1
WRITEI
```

Listing 4.6: Output do teste 3

Utilizando como input os valores 1,2,3,4 determinou-se que o produtório dos valores 1*2*3*4 = 24 o que está correto.

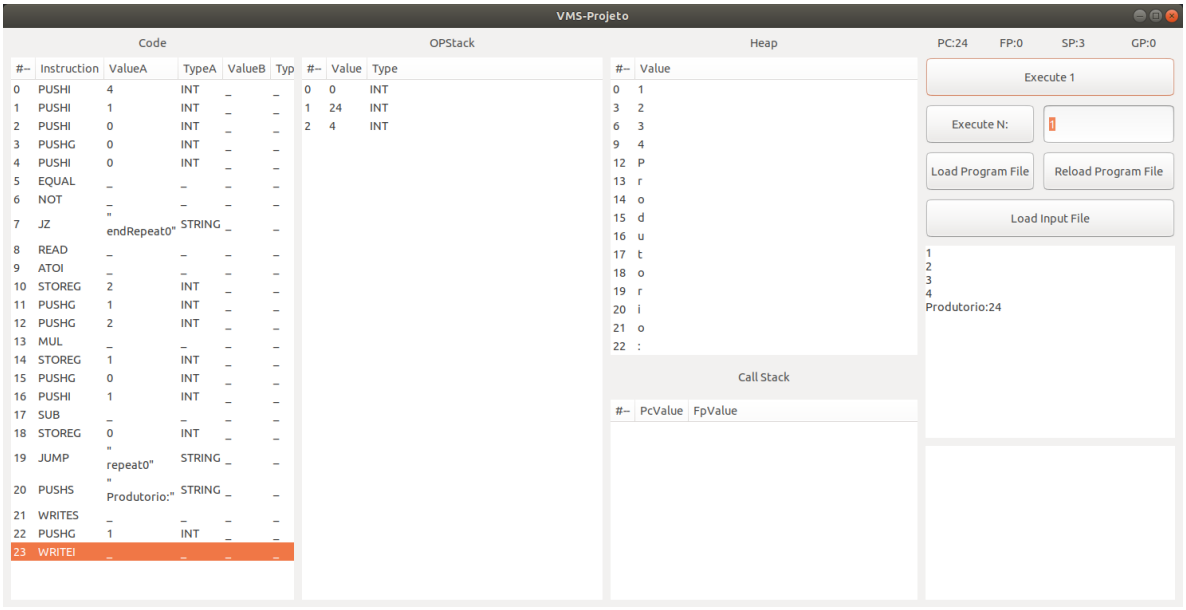

Figura 4.4: Resultado do teste 3 na VM

4.2.4 Números ímpares de um conjunto de números naturais

Para o quarto teste implementou-se um programa que a partir de um conjunto N de valores lidos por input determine quais desses valores são ímpares e o total de números ímpares no conjunto . Implementou-se, assim, o seguinte ficheiro com o respetivo código.

```
1
2  int b
3  int n = read()
4  int c = 0
5
6
7  while (n != 0) { b=read() if((b%2) == 1) { c = c+1 print("Impar:") print(b) } n = n- 1 }
8
9  print("Total:")
10 print(c)
```

Listing 4.7: Teste4

Após a compilação, este ficheiro devolveu o seguinte resultado output com o código máquina. compilador

```
1 PUSHI 0
2 PUSHI 0
3 READ
4 ATOI
5 STOREG 1
```

```
6  PUSHI 0
7
8  repeat0:
9  PUSHG 1
10 PUSHI 0
11 EQUAL
12 NOT
13 jz endRepeat0
14 READ
15 ATOI
16 STOREG 0
17 ifstmnt0:
18 PUSHG 0
19 PUSHI 2
20 MOD
21 PUSHI 1
22 EQUAL
23 jz endif0
24 PUSHG 2
25 PUSHI 1
26 ADD
27 STOREG 2
28 PUSHS "Impar:"
29 WRITES
30 PUSHG 0
31 WRITEI
32 endif0:
33 PUSHG 1
34 PUSHI 1
35 SUB
36 STOREG 1
37 jump repeat0
38 endRepeat0:
39 PUSHS "Total:"
40 WRITES
41 PUSHG 2
42 WRITEI
```

Listing 4.8: Output do teste 4

Utilizando como input 3 valores : 2, 4 e 1 . Calculou-se que o único valor ímpar era o 1, o que está correto.

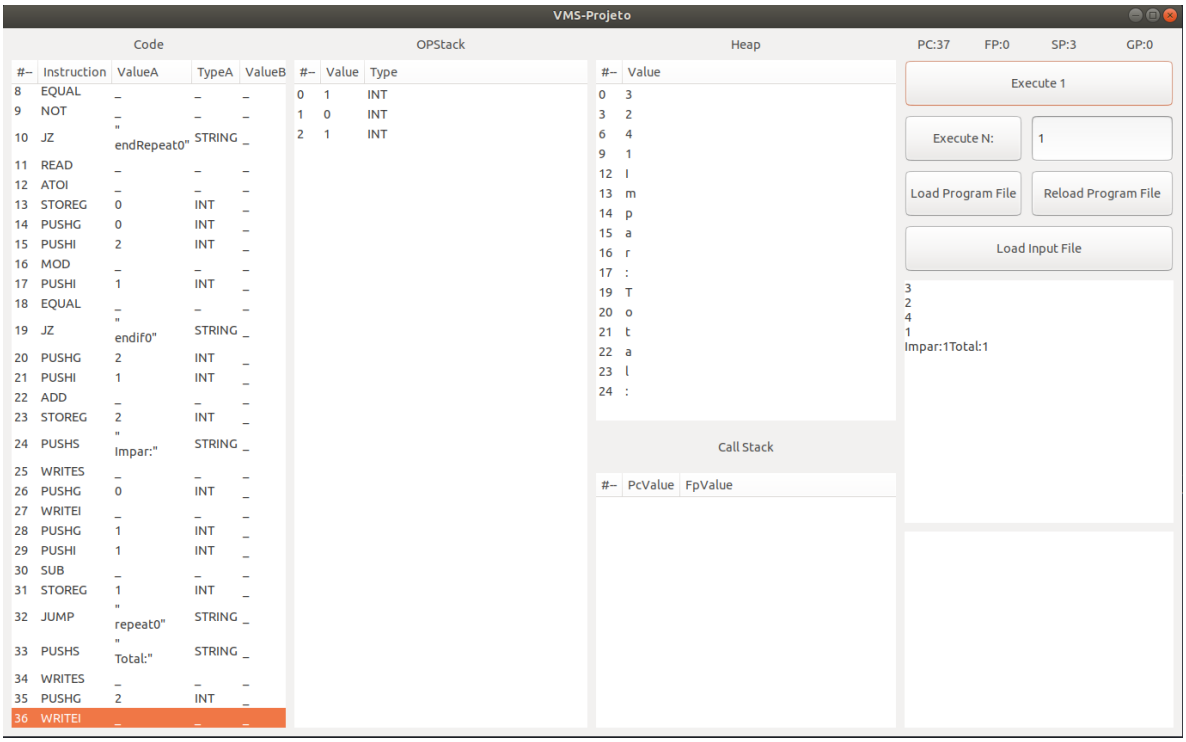

Figura 4.5: Resultado do teste 4 na VM

4.2.5 Definir função potencia

Para o último teste implementou-se um programa onde se definiu um subprograma, potencia(), que começa por ler um valor de base B e um valor de expoente E a partir do input e determina o valor de B elevado a E Implementou-se, assim, o seguinte ficheiro com o respetivo código.

```
1
2 int a = potencia()
3 print("Resultado:")
4 print(a)
5
6 potencia() { int b = read() int e = read() int r=1 while(e!=0) { r = r*b e = e- 1 } return r
      }
```

Listing 4.9: Teste5

Após a compilação, este ficheiro devolveu o seguinte resultado output com o código máquina. compilador

```
1 PUSHI 0
2 PUSHI 0
3 jump subProgram0
4 final0:
5 PUSHG 1
```

```
STOREG 0
PUSHS "Resultado:"
WRITES
PUSHG 0
WRITEI
jump endFunction
subProgram0:
PUSHI 0
READ
ATOI
STOREG 2
PUSHI 0
READ
ATOI
STOREG 3
PUSHI 1

repeat0:
PUSHG 3
PUSHI 0
EQUAL
NOT
jz endRepeat0
PUSHG 4
PUSHG 2
MUL
STOREG 4
PUSHG 3
PUSHI 1
SUB
STOREG 3
jump repeat0
endRepeat0:
PUSHG 4
STOREG 1
jump final0
endProgram0:
endFunction:
```

Listing 4.10: Output do teste 5

Utilizando como input a base 2 e o expoente 4 o resultado calculado é 16 sendo este valor equivalente a 2*2*2*2 = 16 o que está correto.

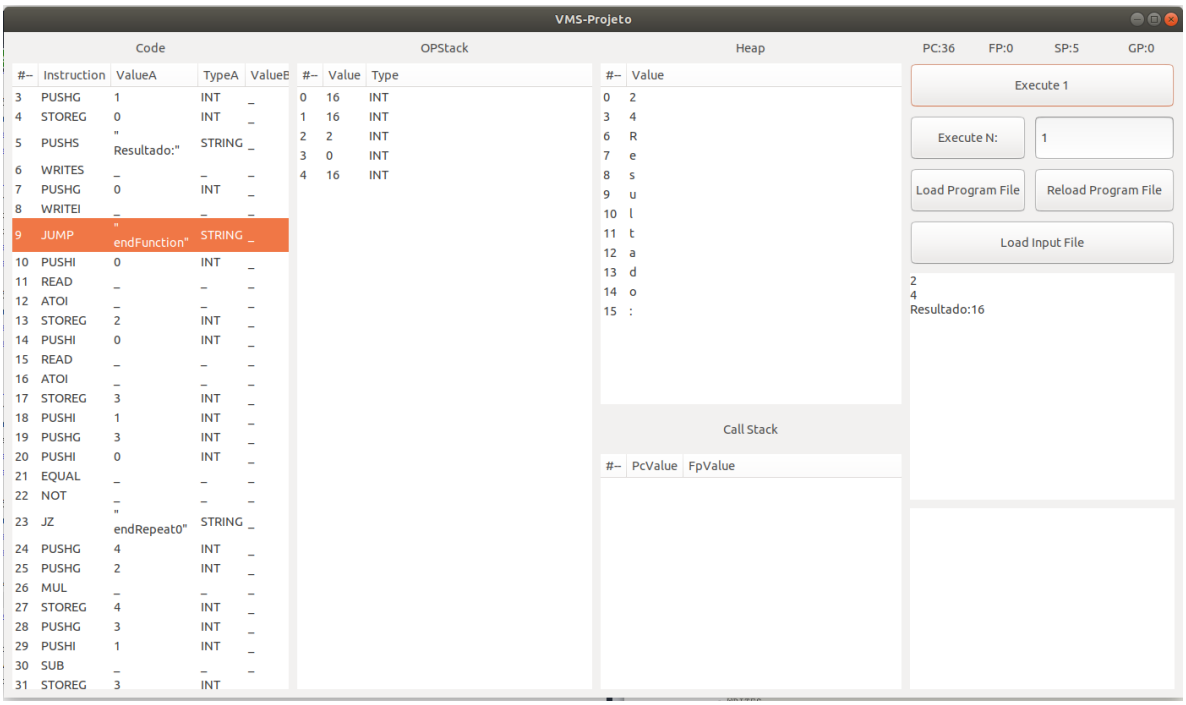

Figura 4.6: Resultado do teste 5 na VM

4.3 Decisões e Problemas de Implementação

Durante a realização do trabalho surgiram alguns casos que necessitavam de ser tratados como exceções à norma, tais como, atribuições de variáveis sem que estas tenham sido declaradas primeiramente e divisões por 0. Para estes casos decidiu-se não aplicar nenhuma ação à respetiva produção, uma vez que como se tratam de casos excepcionais impediam o utilizador de obter um resultado apropriado e correto. Assim, caso esses casos sejam detetados, colocamos em p[0] uma mensagem de erro que será, posteriormente, escrita no ficheiro de output. Este ficheiro ao ficar com erros não pode ser lido pela máquina virtual e com isso o erro ficar controlado.

No que toca às decisões tomadas na implementação, houve algumas que merecem destaque. Entre as diversas opções para a implementação das estruturas de dados, optamos pelos dicionários como forma de armazenar e associar os valores das variáveis à sua respetiva posição na stack devido à sua simplicidade e fácil manuseamento.

Capítulo 5

Conclusão

Dada por concluída a realização do trabalho prático, consideramos boa prática fazer uma apreciação crítica realçando não só os aspetos positivos como também as dificuldades que surgiram e o modo como estas foram colmatadas.

No que diz respeito aos pontos fortes, destacamos o correto funcionamento do compilador implementado que suporta diversas instruções imperativas e também a operacionalidade da funcionalidade adicional, que permite a definição e invocação de subprogramas.

Durante a realização deste trabalho surgiram algumas dificuldades, dos quais destacamos a implementação correta da funcionalidade adicional. Apesar de ter sido ultrapassada, convinha mencionar que esta funcionalidade adicional só funciona corretamente caso haja invocação a uma única função. Mais do que isso e a compilação deixa de ter um funcionamento correto. Isto poderia ser resolvido no futuro assim como a implementação de expressões utilizando *floats* .

Desta forma, pretendemos explicitar que a realização deste projeto foi um aspeto essencial para aprimorar e melhor cimentar os conhecimentos sobre escrita de gramáticas independentes de contexto e tradutoras. Para além disso, uma vez que grande parte das dificuldades foram ultrapassadas e o programa está operacional concluímos que o balanço do resultado final foi positivo.

Apêndice A

Código do Programa

A.1 Programa LEX

```
# calculadora_lex.py
#
# Trabalho Pr tico 2
#

import ply.lex as lex

tokens = ['num','id','INT','IF','EQ','DIF','AND', 'OR','SUPEQ','INFEQ', 'ELSE', 'WHILE','
    READ','PRINT','NOME', 'RETURN', 'STRING']

literals = ['(', ')', '+', '-', '*', '/', '=', '<', '>', '{', '}', '!','[',']', '%', '"']

t_EQ = r'=='
t_DIF = r'!='
t_AND = r'&&'
t_OR = r'\|\|'
t_SUPEQ = r'>='
t_INFEQ = r'<='


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

def t_RETURN(t):
    r'return'
    return t

def t_READ(t):
    r'read\(\)'
    return t
```

```
31
32 def t_WHILE(t):
33     r'while'
34     return t
35
36 def t_INT(t):
37     r'int'
38     return t
39
40 def t_IF(t):
41     r'if'
42     return t
43
44 def t_ELSE(t):
45     r'else'
46     return t
47
48 def t_NOME(t):
49     r'[A-Za-z]+\(\)'
50     return t
51
52 def t_STRING(t):
53     r'\"[A-Za-z]+(:)?\"'
54     return t
55
56 def t_id(t):
57     r'[a-z]'
58     return t
59
60 def t_num(t):
61     r'(\-)?\d+'
62     return t
63
64
65
66 t_ignore = " \t\n\r"
67
68 # se tiver error a nivel lexico
69 def t_error(t):
70     print('Carater ilegal: ', t.value[0]) # acusamos caracter
71     t.lexer.skip(1) # e avancamos
72
73 #build the lexer
74 lexer = lex.lex()
```

Listing A.1: Programa em Python - LEX

A.2 Programa YACC

```
# calculadora_yacc.py
#
# Trabalho Pr tico 2

import ply.yacc as yacc
import sys
#get the token map form the lexer. this is important
from calculadora_lex import tokens
from calculadora_lex import literals

# Production rules -> Regras Producao

f = open("output.txt","w")


def p_Programa(p):
    "Programa : Declaracoes Comandos Funcoes"
    p[0] = p[1] + p[2] + p[3]


def p_Funcoes_Empty(p):
    "Funcoes : "
    p[0] = ""

def p_Funcoes_Funcoes(p):
   "Funcoes : Funcoes Funcao"
   p[0] = p[1] + "jump endFunction\n" + p[2] + "endFunction:\n"


def p_Declaracoes_Empty(p):
    "Declaracoes :  "
    p[0] = ""

def p_Declaracoes_Declaracao(p):
    "Declaracoes : Declaracoes Declaracao"
    p[0] = p[1] + p[2]


def p_Comandos_empty(p):
    "Comandos : "
    p[0] = ""

def p_Comandos_Comando(p):
    "Comandos : Comandos Comando"
    p[0] = p[1] + p[2]


def p_Comando_Atribuicao(p):
```

```
49      "Comando : Atribuicao"
50      p[0] = p[1]
51
52  def p_Comando_Condicao(p):
53      "Comando : Condicao"
54      p[0] = p[1]
55
56  def p_Comando_While(p):
57      "Comando : While"
58      p[0] = p[1]
59
60  def p_Comando_Print(p):
61     "Comando : Print"
62     p[0] = p[1]
63
64
65  # DECLARAR vari veis do tipo inteiro
        -------------------------------------------------------
66  # regra de produ ao para a leitura
67
68  def p_Declaracao_id(p):
69      "Declaracao : INT id"
70      n = len(p.parser.registers)
71      p.parser.registers.update({p[2]: n})
72      p[0] = "PUSHI 0" + "\n"
73
74  def p_Declaracao_atrib(p):
75      "Declaracao : INT id '=' Exp"
76      n = len(p.parser.registers)
77      p.parser.registers.update({p[2]: n})
78      p[0] = p[4] + "\n"
79
80  def p_Declaracao_read(p):
81      "Declaracao : INT id '=' READ"
82      n = len(p.parser.registers)
83      p.parser.registers.update({p[2]: n})
84      p[0] = "PUSHI 0\n" + "READ\n" + "ATOI\n" + "STOREG " + str(n) + "\n"
85
86
87  def p_Declaracao_funcao(p):
88      "Declaracao : INT id '=' NOME"
89
90      n = len(p.parser.registers)
91      p.parser.registers.update({p[2]:n})
92
93      prog = len(p.parser.funcao)
94      p.parser.funcao.update({p[4]: prog})
95
96      tam = len(p.parser.registers)
97      p.parser.registers.update({p[4]:tam})
```

```
98
99      p[0] = "PUSHI 0\n" + "PUSHI 0\n" + "jump subProgram" + str(prog) +"\n" + "final" + str(
        prog) +":\n" +"PUSHG " + str(tam) +"\n" + "STOREG " + str(n) + "\n"
100
101 # ATRIBUICAO  ------------------------------------------------------
102
103 # regra de produ ao para a atribuicao do valor de expressoes numericas a variaveis
104 def p_Atribuicao(p):
105     "Atribuicao : id '=' Exp "
106     n = p.parser.registers.get(p[1])
107     if(n==None):
108         p[0] = "ERRO: ID sem definicao\n"
109     else:
110         p[0] = p[3] + "STOREG " + str(n) + "\n"
111
112 def p_Atribuicao_read(p):
113     "Atribuicao : id '=' READ "
114     n = p.parser.registers.get(p[1])
115     if(n==None):
116         p[0] = "ERRO: ID sem defenicao\n"
117         p.parser.s
118     else:
119         p[0] = "READ\n" + "ATOI\n" + "STOREG " + str(n) + "\n"
120
121 def p_Atribuicao_funcao(p):
122     "Atribuicao : id '=' NOME"
123
124     n = p.parser.registers.get(p[1])
125
126     if(n==None):
127         p[0] = "ERRO: ID sem defenicao\n"
128     else:
129         prog = len(p.parser.funcao)
130         p.parser.funcao.update({p[3]:prog})
131
132         tam = len(p.parser.registers)
133         p.parser.registers.update({p[3]:tam})
134
135         p[0] = "PUSHI 0\n" + "jump subProgram" + str(prog) +"\n" + "final" + str(prog) +":\n
    " + "PUSHG " + str(tam) + "\n" + "STOREG " + str(n) + "\n"
136
137 # DECLARAR funcoes de retorno de inteiros ----------------------------------------
138 def p_Funcao(p):
139     "Funcao : NOME '{' Programa RETURN Exp '}' "
140
141     n = p.parser.registers.get(p[1])
142     prog = p.parser.funcao.get(p[1])
143
144     p[0] = "subProgram" + str(prog) +":\n" + p[3] + p[5] + "STOREG " + str(n) + "\n" + "jump
      final" + str(prog) + "\n" + "endProgram" + str(prog) +":\n"
```

```
145
146 # PRINT de uma express o ----------------------------------------
147 def p_Print(p):
148     "Print : PRINT '(' Exp ')'"
149     #string = '"\n"'
150     p[0] = p[3] + "WRITEI\n"
151
152 def p_Print_string(p):
153     "Print : PRINT '(' STRING ')'"
154     #string = '"\n"'
155     p[0] = "PUSHS " + p[3] + "\nWRITES\n"
156
157 # CONDICAO com if ----------------------------------------
158 def p_Condicao_If(p):
159     "Condicao : IF '(' Comparacoes ')' Corpo"
160     n = p.parser.statement
161     p.parser.statement = n + 1
162     p[0] = "ifstmnt" + str(n) + ":\n" + p[3] + "jz endif" + str(n) +"\n" + p[5] + "endif" +
        str(n) + ":\n"
163
164 def p_Condicao_IfElse(p):
165     "Condicao : IF '(' Comparacoes ')' Corpo ELSE Corpo"
166     n = p.parser.statement
167     p.parser.statement = n + 1
168     p[0] = "ifstmnt" + str(n) + ":\n" + p[3] + "jz endif" + str(n) +"\n" + p[5] + "jump end"
         + str(n) + "\n" + "endif" + str(n) + ":\n" + p[7] + "end" + str(n) + ":\n"
169
170
171 # WHILE ----------------------------------------
172 def p_While(p):
173     "While : WHILE '(' Comparacoes ')' Corpo"
174     n = p.parser.loop
175     p.parser.loop = n + 1
176     p[0] = "repeat" + str(n) + ":\n" + p[3] + "jz endRepeat" + str(n) +"\n" + p[5] + "jump
        repeat" + str(n) + "\n" + "endRepeat" + str(n) + ":\n"
177
178
179 # CORPO ----------------------------------------
180 def p_Corpo_Comando(p):
181     "Corpo : Comando"
182     p[0] = p[1]
183
184 def p_Corpo_Comandos(p):
185     "Corpo : '{' Comandos '}'"
186     p[0] = p[2]
187
188
189 # COMPARACOES ----------------------------------------
190 def p_Comparacoes_Conj(p):
191     "Comparacoes : Comparacao AND Comparacao"
```

```
192     p[0] = p[1] + p[3] + "MUL\n"
193
194 def p_Comparacoes_Disj(p):
195     "Comparacoes : Comparacao OR Comparacao"
196     p[0] = p[1] + p[3] + "ADD\n" + p[1] + p[3] + "MUL\n" + "SUB\n"
197
198 def p_Comparacoes_Comparacao(p):
199     "Comparacoes : Comparacao"
200     p[0] = p[1]
201
202 def p_Comparacoes_NotComparacao(p):
203     "Comparacoes : '!' Comparacao"
204     p[0] = p[2] + "NOT\n"
205
206 # COMPARACAO ----------------------------------------
207 def p_Comparacao_Simples(p):
208     "Comparacao : Factor SimbRelacional Factor"
209     p[0] = p[1] + p[3] + p[2]
210
211 def p_Comparacao_Composta(p):
212     "Comparacao : '(' Comparacoes ')'"
213     p[0] = p[2]
214
215 # SIMBOLOS RELACIONAIS ----------------------------------------
216 def p_SimbRelacional_EQ(p):
217     "SimbRelacional : EQ"
218     p[0] = "EQUAL\n"
219
220 def p_SimbRelacional_DIF(p):
221     "SimbRelacional : DIF"
222     p[0] = "EQUAL\nNOT\n"
223
224 def p_SimbRelacional_INF(p):
225     "SimbRelacional : '<'"
226     p[0] = "INF\n"
227
228 def p_SimbRelacional_INFEQ(p):
229     "SimbRelacional : INFEQ"
230     p[0] = "INFEQ\n"
231
232 def p_SimbRelacional_SUPEQ(p):
233     "SimbRelacional : SUPEQ"
234     p[0] = "SUPEQ\n"
235
236 def p_SimbRelacional_SUP(p):
237     "SimbRelacional : '>'"
238     p[0] = "SUP\n"
239
240 # EXP ----------------------------------------
241 # regra de produ ao para um termo
```

```
242 def p_Exp_termo(p):
243     "Exp : Termo"
244     p[0] = p[1]
245 
246 # regra de produ ao para uma opera ao de soma
247 def p_Exp_add(p):
248     "Exp : Exp '+' Termo"
249     p[0] = p[1] + p[3] + "ADD\n"
250 
251 # regra de produ ao para uma opera ao de subtra ao
252 def p_Exp_sub(p):
253     "Exp : Exp '-' Termo"
254     p[0] = p[1] + p[3] + "SUB\n"
255 
256 # TERMO ----------------------------------------
257 # regra de produ ao para uma opera ao de multiplica ao
258 
259 def p_Termo_mod(p):
260     "Termo : Termo '%' Factor"
261     p[0] = p[1] + p[3] + "MOD\n"
262 
263 def p_Termo_mul(p):
264     "Termo : Termo '*' Factor"
265     p[0] = p[1] + p[3] + "MUL\n"
266 
267 # regra de produ ao para uma opera ao de divisao
268 def p_Termo_div(p):
269     "Termo : Termo '/' Factor"
270     if (p[3]!=0):
271         p[0] = p[1] + p[3] + "DIV\n"
272     else:
273         print("Erro: Divisao por 0, a continuar com 0")
274         p[0] = "ERRO"
275 
276 # regra de produ ao para um factor
277 def p_Termo_factor(p):
278     "Termo : Factor"
279     p[0] = p[1]
280 
281 # FACTOR ---------------------------------------
282 # regra de produ ao para um group de um factor
283 def p_Factor_group(p):
284     "Factor : '(' Exp ')'"
285     p[0] = p[2]
286 
287 # regra de produ ao para um num de um factor
288 def p_Factor_num(p):
289     "Factor : num"
290     p[0] = "PUSHI " + p[1] + "\n"
291 
```

```
292 # regra de produ ao para um id de um factor
293 def p_Factor_id(p):
294     "Factor : id"
295     n = p.parser.registers.get(p[1])
296     p[0] = "PUSHG " + str(n) + "\n"
297 
298 # TRATAMENTO DE ERROS ----------------------------------------
299 # Se nao cai nas regras anteriores, cai nesta regra
300 def p_error(p):
301     print('Erro sintatico: ', p)
302     parser.success = False
303 
304 # Build the parser
305 parser = yacc.yacc()
306 
307 # My state -> manuten  o do estado da stack
308 #durante a execu  o do programa
309 parser.registers = {}
310 parser.funcao = {}
311 parser.statement = 0
312 parser.loop = 0
313 
314 # Read line from input and parse it
315 for linha in sys.stdin:
316     parser.success = True
317     res = parser.parse(linha)
318     if parser.success:
319         print(res)
320         f.write(res)
321 
322     else:
323         print("Frase invalida...Corrija e tente novamente!")
324 f.close()
```

Listing A.2: Programa em Python - YACC