Processamento de Linguagens

Trabalho Prático Nº2

Relatório de Desenvolvimento G57

Júlio Alves	Benjamim Coelho	Henrique Neto
(A89468)	(A89616)	(A89618)

5 de dezembro de 2023

Resumo

Neste relatório vamos abordar como ultrapassamos os desafios que encontrámos na resolução do trabalho prático nº2, cujo objetivo é desenvolver processadores de linguagens segundo o método de tradução gerida pela sintaxe, a partir de uma gramática tradutora e desenvolver um compilador gerando código para uma máquina de stack virtual. Para o desenvolvimento deste trabalho prático, iremos usar a máquina de stack virtual *VM, Virtual Machine* e a biblioteca *PLY* do *Python*, mais concretamente o *YACC* e o *LEX*.

Capítulo 1

Enunciado

Neste projeto pretende-se definir uma linguagem de programação imperativa simples, a nosso gosto. Essa linguagem terá de permitir:

- declarar variáveis atómicas do tipo inteiro, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas.
- efetuar instruções algorítmicas básicas como a atribuição do valor de expressões numéricas a variáveis.
- ler do standard input e escrever no standard output.
- efetuar instruções condicionais para controlo do fluxo de execução.
- efetuar instruções cíclicas para controlo do fluxo de execução, permitindo o seu aninhamento. Note que deve implementar pelo menos o ciclo while-do, repeat-until ou for-do conforme o Número do seu Grupo módulo 3 seja 0, 1 ou 2. Adicionalmente deve ainda suportar, à nossa escolha, uma das duas funcionalidades seguintes:
- declarar e manusear variáveis estruturadas do tipo array (a 1 ou 2 dimensões) de inteiros, em relacão 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.

Como é da praxe neste tipo de linguagens, as variáveis deverão ser declaradas no início do programa e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explicitado, o valor da variável após a declaração é 0 (zero). Temos de desenvolver, então, um compilador para esta linguagem com base na GIC criada acima e com recurso aos módulos Yacc/ Lex do PLY/Python. O compilador deve gerar pseudo-código, Assembly da Máquina Virtual VM.
Devemos também preparar um conjunto de testes (programas-fonte escritos na nossa linguagem) e mostrar 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 nossa escolha acima:

- ler 4 números e dizer se podem ser os lados de um quadrado.
- ler um inteiro N, depois ler N números e escrever o menor deles.
- ler N (constante do programa) números e calcular e imprimir o seu produtório.
- contar e imprimir os números impares de uma sequência de números naturais.

- ler e armazenar N números num array; imprimir os valores por ordem inversa.
- invocar e usar num programa seu uma função 'potencia()', que começa por ler do input a base B e o expoente E e retorna o valor BE.

Capítulo 2

Linguagem de programação imperativa

2.1 Declaração de variáveis

Na nossa linguagem existem 3 tipos de variáveis, sendo elas, um inteiro, um array de inteiros unidimensional e um array de inteiros bidimensional.

Para declarar as mesmas, faz-se da seguinte forma:

```
begindeclares
    arrayInt exemplo[5] - declaração de um array de inteiros unidimensional
    arrayInt exemplo[5][5] - declaração de um array de inteiros bidimensional
    int a - declaração de um inteiro
enddeclares
```

Decidimos ter as palavras "begindeclares"e "enddeclares"para conseguirmos saber mais facilmente quando as declarações acabam. Deste modo, os programas ficam mais organizados e com uma melhor estrutura. Para além disso esta decisão permite ao compilador saber que quando a palavra "enddeclares"aparece, ele pode escrever a instrução "START"da vm.

2.2 Atribuição de valores às variáveis

Para atribuir um valor a uma variável é necessário fazer

```
atr a = 5
atr exemplo[1] = 5
atr exemplo[1][2] = 5
```

Decidímos utilizar a palavra "atr", diminutivo para atribuir, para facilitar o reconhecimento de atribuições, uma vez que todas as operações são uma sequência de carateres. Deste modo, ao verificar que a linha começa por atr, o compilador sabe que a seguir vem uma atribuição.

2.3 Ler no standard input e escrever no standard output

```
read a
print a
prints "Sucesso!"
```

Decidimos tornar a leitura do standard input e escrita no standard output o mais simples possível, utilizando para tal as palavras "read"para ler um valor para uma variável, a palavra "print"para escrever o valor da variável no ecrã e a palavra "prints"(print string) para escrever uma string no ecrã.

2.4 Instruções condicionais

```
if(a>2)
    prints "a>2"
else
    prints "a<=2"
endif

if (a>2 or b>2)
    print "a>2 or b>2"
    else
endif

if (a>2 or (b>2 and a<2))
    print "math is fun!"
    else
endif

if (not a>2)
    print "negação de condições!"
    else
endif
```

As nossas estruturas condicionais são baseadas nas estruturas condicionais da linguagem C, com duas exceções:

- endif - palavra utilizada para o compilador reconhecer o fim da estrutura condicional e saber que tem de meter uma label nesse mesmo sítio.
- else - ao contrário do C, as nossas estruturas condicionais têm sempre de ter um else, mesmo que o corpo deste seja vazio. Isto deve-se ao manuseamento das labels por parte do compilador.

2.5 Instruções cíclicas

Decidimos implementar todos os tipos de loops sugeridos: "while do", "repeat until"e "for do".

```
while(n>0) do
        read input
        if (input<min)
                atr min=input
        endif
        atr n = n - 1
endwhile
```

```
repeat
        read input
        atr produtorio = produtorio * input
        atr N= N - 1
until(N<1)


for(atr i=9;i>=0;atr i= i - 1)do
        print teste[i]
endfor
```

Mais uma vez, à exceção do "repeat until", cada loop tem uma palavra que delimita o fim do seu corpo: "endwhile"e "endfor". Esta decisão surgiu com a necessidade do compilador ser capaz de suportar loops encadeados e gerir corretamente as labels de cada um nas instruções assembly. O loop repeat until não necessita de uma palavra extra pois o "until"já funciona para esse efeito.

2.6 Arrays

```
begindeclares
    arrayInt uniArray[10]
    arrayInt biArray[10][10]
    int a=0
enddeclares

atr uniArray[1] = 5
atr uniArray[a] = 3
atr biArray[1][uniArray[1]] = 200
```

Mais uma vez, a nossa sintaxe segue uma abordagem muito semelhante à da linguagem C, em que, para declarar e aceder a arrays (1 ou 2 dimensões), basta especificar o seu tamanho dentro de parêntesis retos. É importante referir que podemos aceder a valores dos arrays utilizando como índices inteiros ou outras variáveis. Esta funcionalidade torna-se útil para conseguirmos trabalhar com os índices dos arrays iterativamente.

Capítulo 3

Analisador Léxico

Para a criação do analisador léxico da nossa linguagem recorremos a tokens, literals e reserved. No reserved colocamos todas as palavras que possuem uma função no programa e que não poderão ser utilizadas em qualquer outro contexto. Por exemplo, temos como exemplo de palavras reservadas *int*, *print* e *read*. As palavras que são consideradas reservadas, no entanto, também são consideradas tokens. Para além das palavras reservadas, temos como tokens *VAR*, *NUM*, *STRING* e *SIGNEDNUM*. O *VAR* é um token identificado por pelo menos um caractér inicial, sendo que depois pode ter números. Por exemplo "teste10"é um *VAR* válido mas "10teste"já não será. O *NUM* é, como o nome indica, um número, logo qualquer número (à exceção de casos como em *VAR* em que tem um char à sua esquerda) será captado como um *NUM*. O token *STRING* capta tudo o que esteja entre duas aspas. O *SIGNEDNUM* é uma extensão do *NUM*. na medida em que permite que os números possuam um sinal à sua esquerda, indicando se estes são positivos ou negativo. De salientar que este número terá de estar entre parêntesis, pois o *SIGNEDNUM* capta tudo o que esteja entre parêntesis.

Consideramos como literals todos os símbolos que podem ser utilizados na nossa linguagem, entre eles, os parentesis curvos e retos, os operadores aritméticos e o ponto de exclamação.

Capítulo 4

Gramática Independente de Contexto

4.1 Produções da Gramática implementada

Aqui temos um esquema da gramática independente do contexto que utilizámos como base para o nosso compilador. De seguida passaremos a explicar como implementámos cada produção com o módulo YACC do Ply do Python e o raciocínio por trás de cada decisão.

```
Prod 0 : S' -> Programa
Prod 1 : Programa -> Operacoes
Prod 2 : Operacoes -> Operacoes Operacao
Prod 3 :            | Vazio

Prod 4 : Operacao -> BEGINDECLARES
Prod 5 :           | ENDDECLARES
Prod 6 :           | READ VAR
Prod 7 :           | READ VAR '[' EXP ']'
Prod 8 :           | READ VAR '[' EXP ']' '[' EXP ']'
Prod 9 :           | IF '(' CONDICOES ')'
Prod 10 :          | ENDIF
Prod 11 :          | ELSE Operacoes
Prod 12 :          | REPEAT
Prod 13 :          | UNTIL '(' CONDICOES ')'
Prod 14 :          | FOR1 FOR3
Prod 15 :          | ENDFOR
Prod 16 :          | WHILEI ( CONDICOES ) DO
Prod 17 :          | ENDWHILE
Prod 18 :          | INT VAR
Prod 19 :          | INT VAR '=' EXP
Prod 20 :          | ARRAYINT VAR '[' NUM ']'
Prod 21 :          | ARRAYINT VAR '[' NUM ']' '[' NUM ']'
Prod 22 :          | PRINT EXP
Prod 23 :          | ATR VAR '=' EXP
Prod 24 :          | ATR VAR '[' EXP ']' '=' EXP
Prod 25 :          | ATR VAR '[' EXP ']' '[' EXP ']' '=' EXP
Prod 26 :          | PRINTS STRING


Prod 27 : CONDICOES -> CONDICAO
Prod 28 :            | NOT CONDICOES
Prod 29 :            | '(' CONDICOES ')'
```

```
Prod 30 :                | CONDICOES AND CONDICOES
Prod 31 :                | CONDICOES OR CONDICOES



Prod 32 : CONDICAO -> EXP '=' '=' EXP
Prod 33 :               | EXP '!' '=' EXP
Prod 34 :               | EXP '<' EXP
Prod 35 :               | EXP '>' EXP
Prod 36 :               | EXP '<' '=' EXP
Prod 37 :               | EXP '>' '=' EXP


Prod 38 : EXP -> EXP '+' TERMO
Prod 39 :         | EXP '-' TERMO
Prod 40 :         | TERMO


Prod 41 : TERMO -> TERMO '*' FACTOR
Prod 42 :         | TERMO '/' FACTOR
Prod 43 :         | FACTOR


Prod 44 : FACTOR -> '(' EXP ')'
Prod 45 :           | NUM
Prod 46 :           | '(' SIGNEDNUM ')'
Prod 47 :           | VAR
Prod 48 :           | VAR '[' EXP ']'
Prod 49 :           | VAR '[' EXP ']' '[' EXP ']'

Prod 50 : FOR1 -> FOR '(' Operacao ';'

Prod 51 : FOR3 -> FOR2 Operacao ')' DO

Prod 52 : FOR2 -> CONDICOES ';'

Prod 53 : WHILEI -> WHILE
```

4.2 Gramática da Linguagem

Para gerirmos todas as variáveis do programa que está a ser compilado, o nosso parser tem um dicionário como variável (variables). Neste dicionário, as chaves são os nomes das variáveis declaradas e os valores são listas com todas as informações necessárias para a geração de código assembly da VM. Estas informações são:

- variáveis inteiras : tipo e posição na stack
- arrays de uma dimensão : tipo, posição na stack e comprimento
- arrays de duas dimensões : tipo, posição na stack, comprimento total e comprimento de cada linha

Para além disso, o parser também tem uma variável inteira (stack_position) que indica qual a posição do topo da stack atual. Esta variável é útil para sabermos a posição de cada variável declarada na stack.

Um programa da nossa linguagem imperativa é, no fundo, um conjunto de instruções, que designámos por operações (Regra 1).

```
def p_Programa_Operacoes(p):
    "Programa : Operacoes"
    p[0] = p[1]
```

Por sua vez, Este conjunto de operações pode ser visto como uma lista, pelo que, este conjunto pode ser visto como uma operação seguida de mais operações. Neste caso, de modo a ficar com recursividade à esquerda, podemos ver que um conjunto de operações se reduz a outro conjunto de operações seguido de uma operação (regra 2)

```
def p_Operacoes(p):
    "Operacoes : Operacoes Operacao"
    p[0] = p[1] + p[2]
```

Para termos um caso de paragem na recursividade, uma lista de operações pode, obviamente, ser vazia (regra 3)

```
def p_Operacoes_empty(p):
    "Operacoes : "
    p[0] = ''
```

Como referimos anteriormente, uma operação pode ser qualquer instrução que faz parte da nossa linguagem de programação imperativa. Como é costume neste tipo de linguagens, a primeira coisa a se fazer é declarar todas as variáveis que vão ser utilizadas no programa. Na nossa linguagem precisamos de começar as declarações com a palavra "begindeclares". A seguinte produção traduz esta redução (regra 4):

É importante explicar o que cada operação contém. Sempre que uma operação é reconhecida pela nossa gramática, o resultado armazenado em p[0] é o conjunto das instruções vm necessárias para executar o código reconhecido.

```
def p_Operacao_BeginDeclares(p):
    "Operacao : BEGINDECLARES"
    p[0] = ''
```

De seguida, a nossa gramática tem de ser capaz de reconhecer declarações de variáveis inteiras e arrays de 1 ou 2 dimensões. Começando pelo caso mais simples, um utilizador pode declarar um inteiro sem a inicializar. Para isso, tem de escrever o tipo "int"seguido do nome que pretende associar à mesma. Neste caso, a regra 18, trata desse reconhecimento: O resultado desta produção é um "pushi 0"(assumimos que o valor das variáveis não inicializadas é 0). Para além disso, atualizámos o dicionário de variáveis do nosso parser e incrementámos a variável que aponta para o topo da stack.

```
def p_Operacao_int_zero(p):
    "Operacao : INT VAR"
    p[0] = '    pushi 0\n'
    global stack_position
    parser.variables.update({p[2]: ['int', stack_position]})
    stack_position = stack_position + 1
```

Se o utilizador decidir inicializar a variável enquanto a declara, então a produção 19 é aplicada. Para inicializar uma variável inteira, o utilizador tem de igualar uma expressão à mesma. Uma expressão pode ser tanto uma expressão aritmética como uma expressão algébrica em que as variáveis podem ser variáveis inteiras ou um valor de uma posição válida qualquer de um array. A maneira como a nossa gramática

reconhece estas expressões será explicada em mais detalhe posteriormente. O resultado desta produção será então o resultado produzido pela Produção associada às expressões. É necessário também atualizarmos o dicionário de variáveis e a posição do topo da stack.

```
def p_Operacao_int(p):
    "Operacao : INT VAR '=' EXP"
    p[0] = p[4]
    global stack_position
    parser.variables.update({p[2]: ['int', stack_position]})
    stack_position = stack_position + 1
```

Para inicializar arrays de 1 dimensão, o utilizador tem de escrever o tipo "ArrayInt", seguido do nome e do seu comprimento (inteiro entre parêntesis retos, Regra 20). Para além de atualizar o dicionário de variáveis e o topo da stack, o resultado desta produção terá de ser um "push n"onde n é o comprimento do array introduzido pelo utilizador.

```
def p_Operacao_arrayInt(p):
    "Operacao : ARRAYINT VAR '[' NUM ']'"
    global stack_position
    parser.variables.update({p[2]: ['arrayInt', stack_position, p[4]]})
    stack_position = stack_position + int(p[4])
    p[0] = '    pushn ' + p[4] + '\n'
```

Para inicializar arrays de 2 dimensões, o utilizador tem, para além de escrever tudo o que é necessário para inicializar um array de uma dimensão, a dimensão de cada coluna (inteiro entre parêntesis retos). Para a vm, arrays de 2 dimensões funcionam exatamente da mesma maneira que os de 1 dimensão. Assim, o nosso compilador tem de ser capaz de traduzir operações sobre arrays de duas dimensões para operações de arrays de 1 dimensão que são executáveis na vm. Deste modo, a produção 21 trata de atualizar o dicionário de variáveis, colocando no campo do comprimento total do array, o produto das duas dimensões inseridas pelo utilizador. Para além disso, o resultado é também um "push n", onde n é o produto calculado anteriormente.

```
def p_Operacao_BiArrayInt(p):
    "Operacao : ARRAYINT VAR '[' NUM ']' '[' NUM ']'"
    global stack_position
    aux = int(p[4]) * int(p[7])
    parser.variables.update(
        {p[2]: ['arrayInt', stack_position, str(aux), p[4]]})
    stack_position = stack_position + aux
    p[0] = '    pushn ' + str(aux) + '\n'
```

Uma das operações que a nossa linguagem tem de ser capaz de realizar é de receber input e gerar output. Para receber input, o utilizador pode utilizar a palavra "read"seguida do nome da variável onde quer armazenar este valor. Para proceder ao reconhecimento desta operação, criámos as regras 6,7 e 8, uma para cada tipo de variável. Um aspeto muito importante que é relevante de se explicar é que a nossa linguagem permite aceder às posições dos arrays com qualquer expressão numérica ou algébrica válida.

```
def p_Operacao_Read(p):
    "Operacao : READ VAR"
    global variables
    atributes = parser.variables[p[2]]
    p[0] = '    read\n' + '    atoi\n' + \
        '    storeg ' + str(atributes[1]) + '\n'
```

Como podemos observar o resultado destas produções é sempre um "read"seguido de um "atoi"para converter a string para inteiro, seguido de um store apropriado ao tipo de variável em causa.

```
def p_Operacao_ReadToArray(p):
    "Operacao : READ VAR '[' EXP ']'"
    atributes = parser.variables[p[2]]
    p[0] = '    pushgp\n' + '    pushi ' + \
        str(atributes[1]) + '\n' + '    padd\n' + p[4] + \
        '    read\n' + '    atoi\n' + '    storen\n'
```

Para guardar o valor calculado pelo "atoi"na posição do array, é necessário utilizar a instrução "storen"da vm. Assim, necessitams de calcular o endereço do array na stack, seguido da posição onde vamos inserir o valor, seguido do valor que vamos inserir.

```
def p_Operacao_ReadToBiArray(p):
    "Operacao : READ VAR '[' EXP ']' '[' EXP ']'"
    atributes = parser.variables[p[2]]
    p[0] = '    pushgp\n' + '    pushi ' + str(atributes[1]) +
        '\n' + '    padd\n' + p[4] + str(
        atributes[3]) + '  mul\n' + str(atributes[3]) + '  add\n' +
        '    read\n' + '    atoi\n' + '    storen\n'
```

Em arrays de duas dimensões, calcular a posição onde queremos guardar o valor é um pouco mais complicado. Como podemos observar, nas informações destes arrays temos, para além do comprimento total, o comprimento da cada linha. Assim, tendo o resultado das expressões dos índices, basta multiplicar o primeiro índice pelo comprimento de cada linha deste array e somar o segundo índice. De resto, o processo é exatamente igual ao do array de uma dimensão, bastando utilizar a instrução storen para guardar o valor recebido como input.

Para imprimir o valor de variáveis no ecrã, o utilizador pode escrever a palavra "print"seguida da variável ou expressão aritmética (produção 22). Como, como vai ser visto posteriormente, uma expressão pode ser tanto uma variável como operações sobre inteiros, esta produção faz uso do token EXP para detetar e calcular o valor que o utilizador quer imprimir no ecrã. O resultado desta produção é simplesmente o valor calculado pela expressão, seguido pela instrução "writei"da vm.

```
def p_Operacao_Print_EXP(p):
    "Operacao : PRINT EXP"
    p[0] = p[2] + '    writei\n'
```

Para além de imprimir variáveis, o utilizador pode imprimir texto no ecrã. Para tal, pode fazer uso da operação "prints", que deve ser seguida do texto delimitado por aspas. Para reconhecer esta operação, criámos a produção 26. O resultado é a instrução "pushs"da vm, seguida pelo texto inserido pelo utilzador, finalizando com a instrução "writes".

```
def p_Operacao_PRINTS(p):
    "Operacao : PRINTS STRING "
    p[0] = '    pushs ' + p[2] + '\n    writes\n'
```

Outra operação suportada pela nossa linguagem, é a atribuição de novos valores às variáveis que foram declaradas. Para este efeito, o utilizador começa por escrever a palavra "atr"seguida da variável em questão , o literal '=' e, finalmente, a expressão que calcula o valor desejado. As produções 23,24 e 25 tratam do reconhecimento destas operações. A maneira como os valores das variáveis são atualizados é exatamente a mesma da operação "read"explicada anteriormente, com a diferença de que o valor em vez de ser inserido pelo utilizador como input é calculado através da expressão.

```
def p_Operacao_Atribuir(p):
    "Operacao : ATR VAR '=' EXP"
    atributes = parser.variables[p[2]]
    p[0] = p[4] + '    storeg ' + str(atributes[1]) + '\n'
```

```
def p_Operacao_Atribuir_Array(p):
    "Operacao : ATR VAR '[' EXP ']' '=' EXP"
    atributes = parser.variables[p[2]]
    p[0] = '    pushgp\n' + '    pushi ' + \
        str(atributes[1]) + '\n' + '    padd\n' + p[4] + p[7] + '    storen\n'

def p_Operacao_Atribuir_BiArray(p):
    "Operacao : ATR VAR '[' EXP ']' '[' EXP ']' '=' EXP"
    atributes = parser.variables[p[2]]
    p[0] = '    pushgp\n' + '    pushi ' + str(atributes[1]) + '\n'
    + '    padd\n' + p[4] + \
    '    pushi ' + atributes[3] + '\n' + '    mul\n' + \
        p[7] + '    add\n' + p[10] + '    storen\n'
```

Como referimos anteriormente, uma expressão pode ser uma mistura de operações sobre inteiros com operações sobre variáveis. Para além disto, nas operações aritméticas, existem certos operadores que têm prioridade sobre outros, influenciando o resultado final. Deste modo, criámos as produções 38,39,40,41,42,43,44,45,46,47,4 e 49.

Esta produção tem como objetivo reconhecer a operação de soma entre dois valores. A razão pela qual a soma e a subtração estão neste "nível"da gramática é porque são os operadores aritméticos com menor prioridade pelo que devem ser aplicados por último. Deste modo, estes operadores só são aplicados, após ter havido reduções para EXP e TERMO, ou seja, após as outras operações terem sido concluídas. No caso da soma, o resultado da produção é simplesmente os dois valores calculados por redução, seguidos da instrução "add"da vm.

```
def p_EXP(p):
    "EXP : EXP '+' TERMO"
    p[0] = p[1] + p[3] + '    add\n'
```

Para a produção que detecta a operação de subtração, o resultado é também os dois valores calculados pelas reduções, seguidos da instrução "sub".

```
def p_EXP_sub(p):
    "EXP : EXP '-' TERMO"
    p[0] = p[1] + p[3] + '    sub\n'
```

Ora, para permitir que uma expressão possa também ter as outras operações restantes, é importante termos a seguinte produção que permita que um TERMO seja reduzido a uma expressão, sendo o resultado, o resultado do TERMO.

```
def p_TERMO(p):
    "EXP : TERMO"
    p[0] = p[1]
```

As operações aritméticas multiplicação e divisão têm maior prioridade que a soma e subtração, pelo que o seu reconhecimento é efetuado pelas seguintes produções relativas a TERMO. Para o caso do produto, o resultado é os dois valores calculados pelas reduções seguidos da instrução "mul".

```
def p_TERMO_mul(p):
    "TERMO : TERMO '*' FACTOR"
    p[0] = p[1] + p[3] + '    mul\n'
```

Para a divisão, o resultado é também os dois valores calculados pelas reduções, seguidos pela instrução "div"

```
def p_TERMO_div(p):
    "TERMO : TERMO '/' FACTOR"
    p[0] = p[1] + p[3] + '    div\n'
```

Para permitir que um termo possa ter as operações restantes, temos de ter a seguinte produção que permita que um FACTOR seja reduzido a um TERMO, sendo o resultado o mesmo resultado do FACTOR.

```
def p_FACTOR(p):
    "TERMO : FACTOR"
    p[0] = p[1]
```

Como os parêntesis têm como função dar prioridade máxima à expressão no seu interior, aparecem no "nível"do FACTOR, para garantir que são calculadas em primeiro lugar, uma vez que são reduzidas primeiro. O resultado desta produção será então o resultado da expressão no interior dos parêntesis.

```
def p_FACTOR_group(p):
    "FACTOR : '(' EXP ')'"
    p[0] = p[2]
```

Um FACTOR pode ser um número inteiro qualquer, pelo que a seguinte redução devolve a instrução "pushi"seguida do número em causa.

```
def p_FACTOR_NUM(p):
    "FACTOR : NUM"
    p[0] = '    pushi ' + p[1] + '\n'
```

Como alternativa, o utilizador pode querer escrever números com os sinais positivo e negativo (que são reconhecidos pelo token SIGNEDNUM). O resultado é também a instrução "pushi"seguida do número em causa.

```
def p_FACTOR_SIGNEDNUM(p):
    "FACTOR : SIGNEDNUM"
    p[0] = '    pushi ' + getSignedNum(p[1]) + '\n'
```

Como foi dito anteriormente, uma expressão pode conter variáveis. Assim, a seguinte produção reconhece estes casos e garante que o valor associado às variáveis inteiras é colocado na stack, utilizando para tal a instrução "pushg"seguida da localização da variável na stack.

```
def p_FACTOR_VAR(p):
    "FACTOR : VAR"
    atributes = parser.variables[p[1]]
    p[0] = '    pushg ' + str(atributes[1]) + '\n'
```

Uma variável pode também ser um valor de um array. Deste modo, a produção seguinte calcula a localização do valor em causa e coloca-lo na stack, utilizando a instrução "loadn".

```
def p_FACTOR_ArrayInt(p):
    "FACTOR : VAR '[' EXP ']'"
    atributes = parser.variables[p[1]]
    p[0] = '    pushgp\n' + '    pushi ' + \
        str(atributes[1]) + '\n' + '    padd\n' + p[3] + '\n' + '    loadn\n'
```

Utilizando os mesmos cálculos que foram explicados na operação read para arrays, a produção apresentada a seguir calcula a posição do valor pretendido do array de 2 dimensões e coloca-o na stack, utilizando a instrução "loadn".

```
def p_FACTOR_BiArrayInt(p):
    "FACTOR : VAR '[' EXP ']' '[' EXP ']'"
    atributes = parser.variables[p[1]]
    p[0] = '    pushgp\n' + '    pushi ' + \
        str(atributes[1]) + '\n' + '    padd\n' + p[3] + \
        '    pushi ' + atributes[3] + '\n' + '    mul\n' + \
        p[6] + '    add\n' + '    loadn\n'
```

De forma a tratar as operações condicionais, criamos as seguintes produções 9, 10 e 11. Para iniciar um if, o utilizador terá de escrever if, seguido de condicoes, sendo que estas terão de estar obrigatoriamente entre parêntesis. Para garantir que a vm sabe sempre qual o if para que deve ir, criamos duas variáveis globais, if_counter e if_stack. O inteiro if_counter permite-nos saber quantos IFs já existem no programa e assim conseguimos criar labels utilizando este número, de modo a não repetir nenhuma. A variável if_stack é uma lista que, como o nome indica, é uma stack dos índices dos IFs que aparecem. A necessidade desta stack apareceu com a necessidade de suportar IF's aninhados. Por exemplo, se, no corpo de um IF aparecesse outro IF, o compilador precisa de saber qual o índice que deve atribuir às labels do IF interior, sem perder a informação dos índices do IF exterior para escrever a label do else e do seu fim. Com esta stack, conseguimos fazer push e pop dos índices, garantido que os índices das labels são sempre os referentes ao IF's corretos. Um mecanismo muito semelhante foi utilizado para implementar os ciclos. Neste caso em específico, começaremos por imprimir as condições, que serão explicadas mais à frente, de forma a poder testar se as mesmas são válidas e se se poderá entrar no corpo do if. De seguida imprime-se a instrução para ir para o else correspondente caso essa condição não seja válida.

```
def p_Operacao_BeginIF(p):
    "Operacao : IF '(' CONDICOES ')'"
    global if_counter, if_stack
    push(if_stack, if_counter)
    p[0] = p[3]+'    jz ' + 'else' + str(if_counter) + '\n'
    if_counter += 1
```

Para marcar o fim de um if, criamos a produção 10 que obriga a que o utilizador escreva "endif"no fim do corpo de um if, de forma a sinalizar que o que vem a seguir já não se encontra abrangido pelas regras do if.

```
def p_Operacao_EndIF(p):
    "Operacao : ENDIF"
    global if_stack
    counter = pop(if_stack)
    p[0] = 'endIf' + str(counter) + ':\n'
```

Como para todo o if é necessário um else, criamos a produção 11. Esta operação deverá ser sempre efetuada antes do endif. Fazendo o pop da stack, pegamos no valor que está no topo da stack e que nos indica qual o if em que nos encontramos atualmente. Após isso, imprimimos a instrução de salto para o endIf correspondente e, após o endIf, imprimimos a label do else atual, conjuntamente com as operações indicadas nesse else.

```
def p_Operacao_else(p):
    "Operacao : ELSE Operacoes"
    global if_stack
    counter = top(if_stack)
    p[0] = '    jump ' + 'endIf' + \
        str(counter) + '\n' + 'else' + str(counter) + ':\n' + p[2]
```

Para definir as condições, implementamos as produções 27, 28, 29, 30, 31. Estas são exatamente como indicado nas produções, escrevendo primeiro as condições e no fim escrevendo a operação assembly que lhes corresponde. Dando como exemplo a produção 30.

```
def p_CONDICOES_AND(p):
    "CONDICOES : CONDICOES AND CONDICOES"
    p[0] = p[1] + p[3] + '    mul\n'

def p_CONDICOES(p):
    "CONDICOES : CONDICAO"
    p[0] = p[1]
```

As *CONDICOES* são definidas por *CONDICAO*. Para definir *CONDICAO*, criamos as produções 32 a 37. Estas, assim como as *CONDICOES* são bastante simples, em que começamos por imprimir cada um dos membros da condição, seguido da instrução assembly que lhe corresponde. A título de exemplo, podemos ver por exemplo a produção 32.

```
def p_CONDICAO_Igual(p):
    "CONDICAO : EXP '=' '=' EXP"
    p[0] = p[1] + p[4] + '    equal\n'
```

Definimos 3 tipos de loops, sendo eles os loops repeat-until (produção 12 e 13), while-do (produções 16, 17 e 53) e for-do (produções 14,50,51 e 52).

Um dos problemas que nos apareceu na realização dos loops, nomeadamente do for-do, foi que precisavamos de fazer a atribuição da variável antes de iniciar o ciclo e de a alterar apenas no final do corpo. Para ultrapassar esses obstáculos criamos as seguintes produções:

```
def p_Operacao_For(p):
    "Operacao : FOR1 FOR3"
    p[0] = p[1] + p[2]

def p_FOR1_FOR(p):
    "FOR1 : FOR '(' Operacao ';'"
    global for_counter
    global for_stack
    push(for_stack, for_counter)
    p[0] = p[3] + 'fordo' + str(for_counter) + ':\n'

def p_FOR2(p):
    "FOR2 : CONDICOES ';'"
    global for_counter
    p[0] = p[1] + '    jz endfordo' + str(for_counter) + '\n'
    for_counter = for_counter + 1

def p_FOR3(p):
    "FOR3 : FOR2 Operacao ')' DO"
    global tmp
    tmp = str(p[2])
    p[0] = p[1]
```

Com a criação do *FOR1*, garantimos que a variável tem a sua atribuição antes de o ciclo iniciar e com o *FOR3*, guardamos a operação que modifica a variável sobre a qual estamos a operar numa variável global, permitindo que esta possa ser escrita na produção 15 e garantindo que a variável só será alterada no fim do corpo do ciclo.

```
def p_Operacao_EndFor(p):
    "Operacao : ENDFOR"
    global for_counter, tmp, for_stack
    counter = pop(for_stack)
    p[0] = tmp + '\n' + '    jump fordo' + \
        str(counter) + '\n' + 'endfordo' + str(counter) + ':\n'
```

Para o repeat-until utilizamos um mecanismo semelhante ao if para saber sempre qual o ciclo onde nos encontramos, sendo estas as produções 12 e 13.

```
def p_Operacao_Repeat(p):
    "Operacao : REPEAT"
```

```
    global repeat_counter
    global repeat_stack
    push(repeat_stack, repeat_counter)
    p[0] = 'repeat' + str(repeat_counter) + ':\n'
    repeat_counter += 1


def p_Operacao_Until(p):
    "Operacao : UNTIL '(' CONDICOES ')'"
    global repeat_stack
    global repeat_counter
    counter = pop(repeat_stack)
    p[0] = p[3] + '    jz ' + 'repeat' + \
        str(counter) + '\n' + 'endrepeat' + str(counter) + ':\n'
```

Para o ciclo while-do, tivemos de seguir um caminho semelhante àquele que seguimos para o ciclo for-do, na medida em que necessitavamos que a label fosse escrita antes de qualquer coisa, pelo que por esse motivo criamos a produção 53

```
def p_WHILEI(p):
    "WHILEI : WHILE"
    global while_counter
    global while_stack
    p[0] = 'whiledo' + str(while_counter) + ':\n'
```

Sendo que após tratarmos deste pormenor, a produção do while é uma produção como todas as outras.

```
def p_Operacao_While(p):
    "Operacao : WHILEI '(' CONDICOES ')' DO"
    global while_counter
    global while_stack
    push(while_stack, while_counter)
    p[0] = p[1] + p[3] + '    jz endwhile' + str(while_counter) + '\n'
    while_counter = while_counter + 1
```

Capítulo 5

Testes

Para utilizar o nosso compilador, basta passar como argumento o nome do ficheiro com o código fonte. Em alternativa podemos passar o argumento -help"para verificar outras opções de utilização. Por exemplo, depois do ficheiro de input podemos colocar a opção -s"para correr o compilador em modo silencioso, isto é, sem output do reconhecimento da linguagem.

5.1 Quadrado

Este programa tem como objetivo ler 4 números e dizer se podem ser os lados de um quadrado. Permitiu-nos testar a introdução de input por parte do utilizador, os loops "while do"e as estruturas condicionais.

5.1.1 Código original

```
begindeclares
        int first
        int input
        int counter=0
        int output=1
enddeclares

read first

while(counter<3 and output==1) do
        read input
        if(input!=first)
                atr output=0
        else
        endif
        atr counter= counter + 1
endwhile

if(counter == 3 and output==1)
prints "Podem ser lados de um quadrado"
else
prints "Não podem ser lados de um quadrado"
endif
```

```
print output
```

5.1.2 Código VM

```
  pushi 0
   pushi 0
   pushi 0
   pushi 1
START
   read
   atoi
   storeg 0
whiledo0:
   pushg 2
   pushi 3
   inf
   pushg 3
   pushi 1
   equal
   mul
   jz endwhile0
   read
   atoi
   storeg 1
   pushg 1
   pushg 0
   sub
   jz else0
   pushi 0
   storeg 3
   jump endIf0
else0:
endIf0:
   pushg 2
   pushi 1
   add
   storeg 2
   jump whiledo0
endwhile0:
   pushg 2
   pushi 3
   equal
   pushg 3
   pushi 1
   equal
   mul
   jz else1
   pushs "Podem ser lados de um quadrado"
   writes
   jump endIf1
else1:
   pushs "Não podem ser lados de um quadrado"
```

```
    writes
endIf1:
    pushg 3
    writei
STOP
```

5.1.3 Demonstração execução com interface gráfica

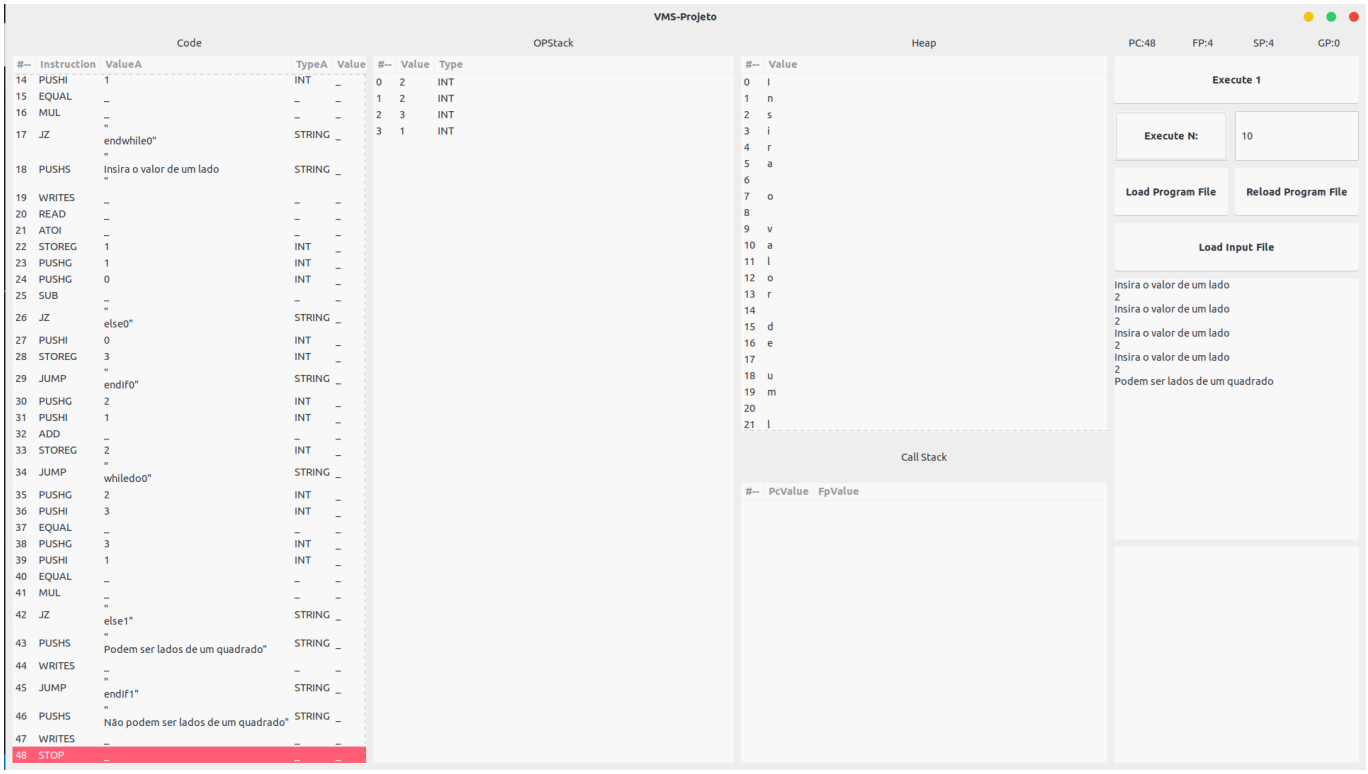

Figura 5.1: Resultado de uma execução em que os inputs dados podem criar um quadrado

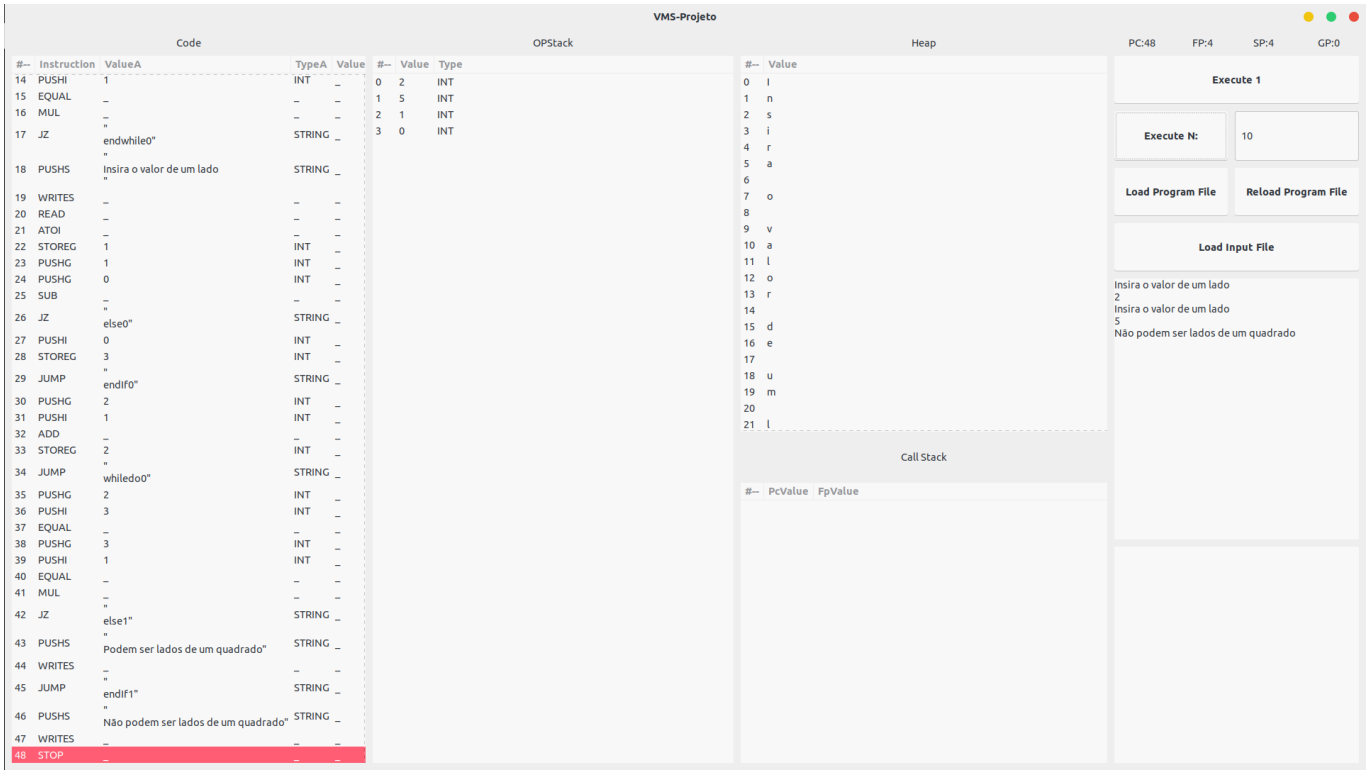

Figura 5.2: Resultado de uma execução em que os inputs dados não podem criar um quadrado

5.2 Impares

Este programa tem como objetivo contar e imprimir os números impares de uma sequência de números naturais. Permitiu-nos testar loops encadeados e o funcionamento do loop for.

5.2.1 Código original

```
begindeclares
        int N
        int input
        int aux
        int total=0
enddeclares

for(atr N=0;N<4;atr N= N + 1)do
        read input
        atr aux = input

        while (aux > 0) do
                atr aux = aux - 2
        endwhile

        if (aux==(-1))
                print input
                atr total = total + 1
```

```
        else
        endif
endfor

print total
```

5.2.2 Código VM

```
    pushi 0
    pushi 0
    pushi 0
    pushi 0
START
    pushi 0
    storeg 0
fordo0:
    pushg 0
    pushi 4
    inf
    jz endfordo0
    read
    atoi
    storeg 1
    pushg 1
    storeg 2
whiledo0:
    pushg 2
    pushi 0
    sup
    jz endwhile0
    pushg 2
    pushi 2
    sub
    storeg 2
    jump whiledo0
endwhile0:
    pushg 2
    pushi -1
    equal
    jz else0
    pushg 1
    writei
    pushg 3
    pushi 1
    add
    storeg 3
    jump endIf0
else0:
endIf0:
    pushg 0
    pushi 1
    add
    storeg 0
```

```
    jump fordo0
endfordo0:
    pushg 3
    writei
STOP
```

5.2.3 Demonstração execução com interface gráfica

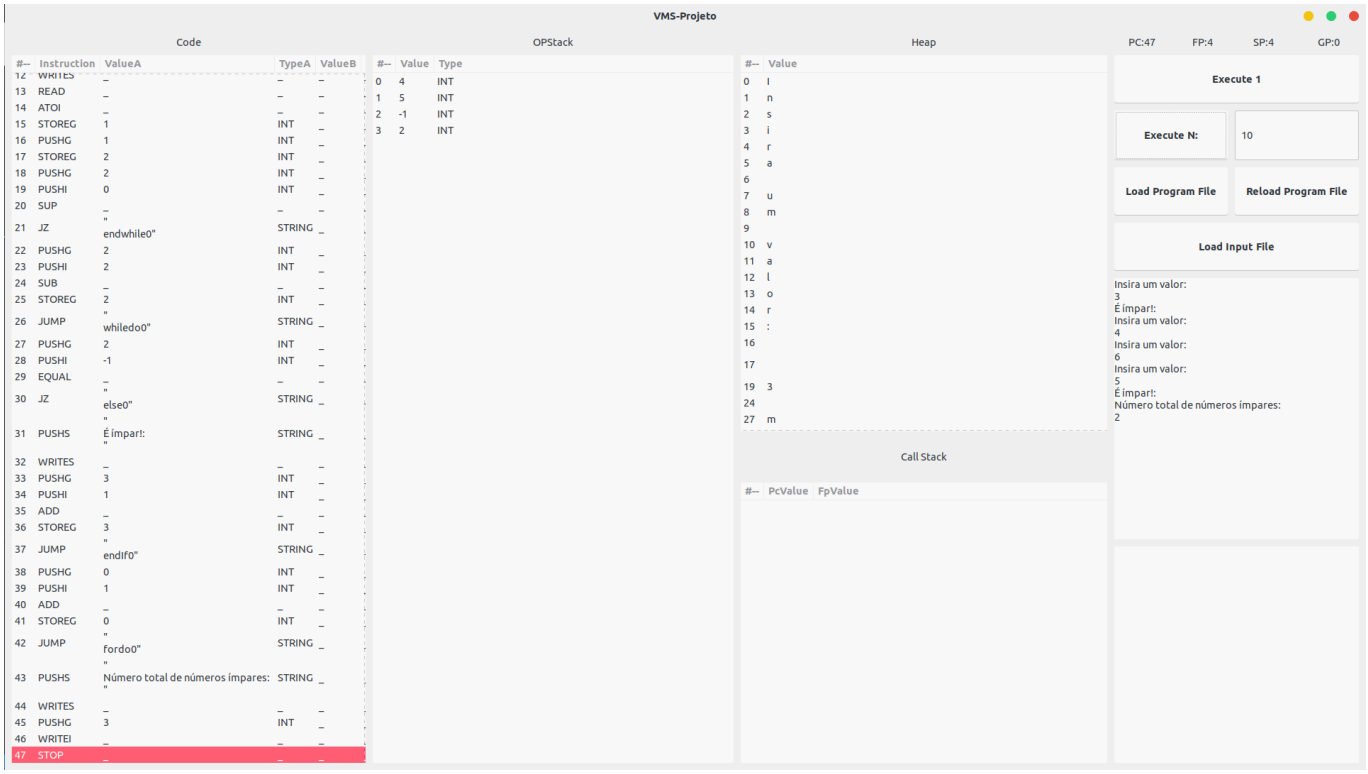

Figura 5.3: Resultado de uma execução de uma sequência de N números naturais e saber se são ímpares

5.3 menorN

Este programa tem como objetivo ler um inteiro N, depois ler N números e escrever o menor deles. Permitiu-nos também testar o funcionamento do loop "while do"e das estruturas condicionais.

5.3.1 Código original

```
begindeclares
        int input
        int min
        int n
enddeclares

read n
```

```
read min
atr n= n - 1

while(n>0) do
        read input
        if (input<min)
                atr min=input
        else
        endif
        atr n = n - 1
endwhile

print min
```

5.3.2 Código VM

```
  pushi 0
   pushi 0
   pushi 0
START
   read
   atoi
   storeg 2
   read
   atoi
   storeg 1
   pushg 2
   pushi 1
   sub
   storeg 2
whiledo0:
   pushg 2
   pushi 0
   sup
   jz endwhile0
   read
   atoi
   storeg 0
   pushg 0
   pushg 1
   inf
   jz else0
   pushg 0
   storeg 1
   jump endIf0
else0:
endIf0:
   pushg 2
   pushi 1
   sub
   storeg 2
   jump whiledo0
```

```
endwhile0:
    pushg 1
    writei
STOP
```

5.3.3 Demonstração execução com interface gráfica

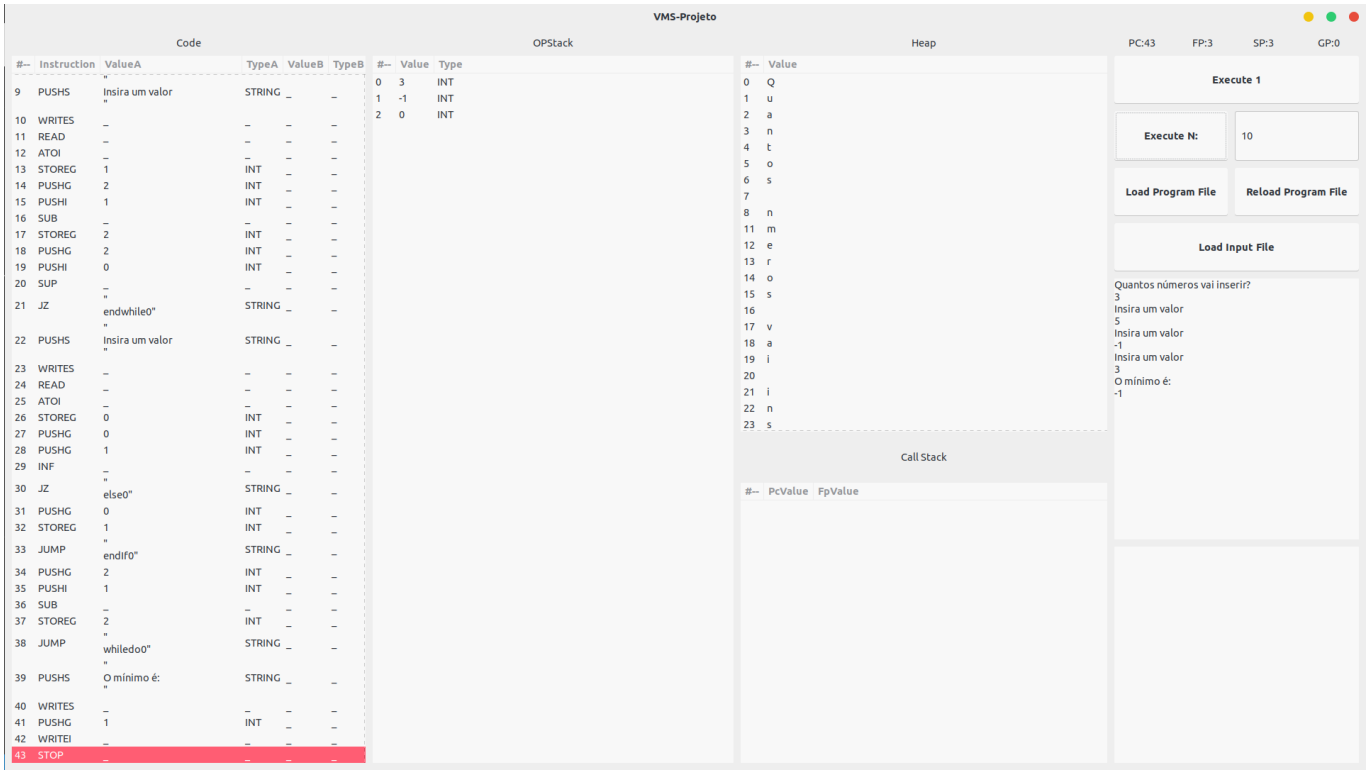

Figura 5.4: Resultado de uma execução de uma sequência de N números e saber qual é o menor

5.4 Array inverso

Este programa tem como objetivo ler e armazenar N números num array e imprimir os valores por ordem inversa. Permitiu-nos assim testar o funcionamento dos arrays e dos loops for.

5.4.1 Código original

```
begindeclares
        int i
        arrayInt teste[10]
        int input
enddeclares

for(atr i=0;i<10;atr i= i + 1)do
        prints "Insira um valor para inserir no array"
        read input
        atr teste[i]=input
```

```
endfor

for(atr i=9;i>=0;atr i= i - 1)do
        print teste[i]
endfor
```

5.4.2 Código VM

```
    pushi 0
    pushn 10
    pushi 0
START
    pushi 0
    storeg 0
fordo0:
    pushg 0
    pushi 10
    inf
    jz endfordo0
    pushs "Insira um valor para inserir no array"
    writes
    read
    atoi
    storeg 11
    pushgp
    pushi 1
    padd
    pushg 0
    pushg 11
    storen
    pushg 0
    pushi 1
    add
    storeg 0

    jump fordo0
endfordo0:
    pushi 9
    storeg 0
fordo1:
    pushg 0
    pushi 0
    supeq
    jz endfordo1
    pushgp
    pushi 1
    padd
    pushg 0

    loadn
    writei
    pushg 0
    pushi 1
```

```
sub
storeg 0

jump fordo1
```

5.4.3 Demonstração execução com interface gráfica

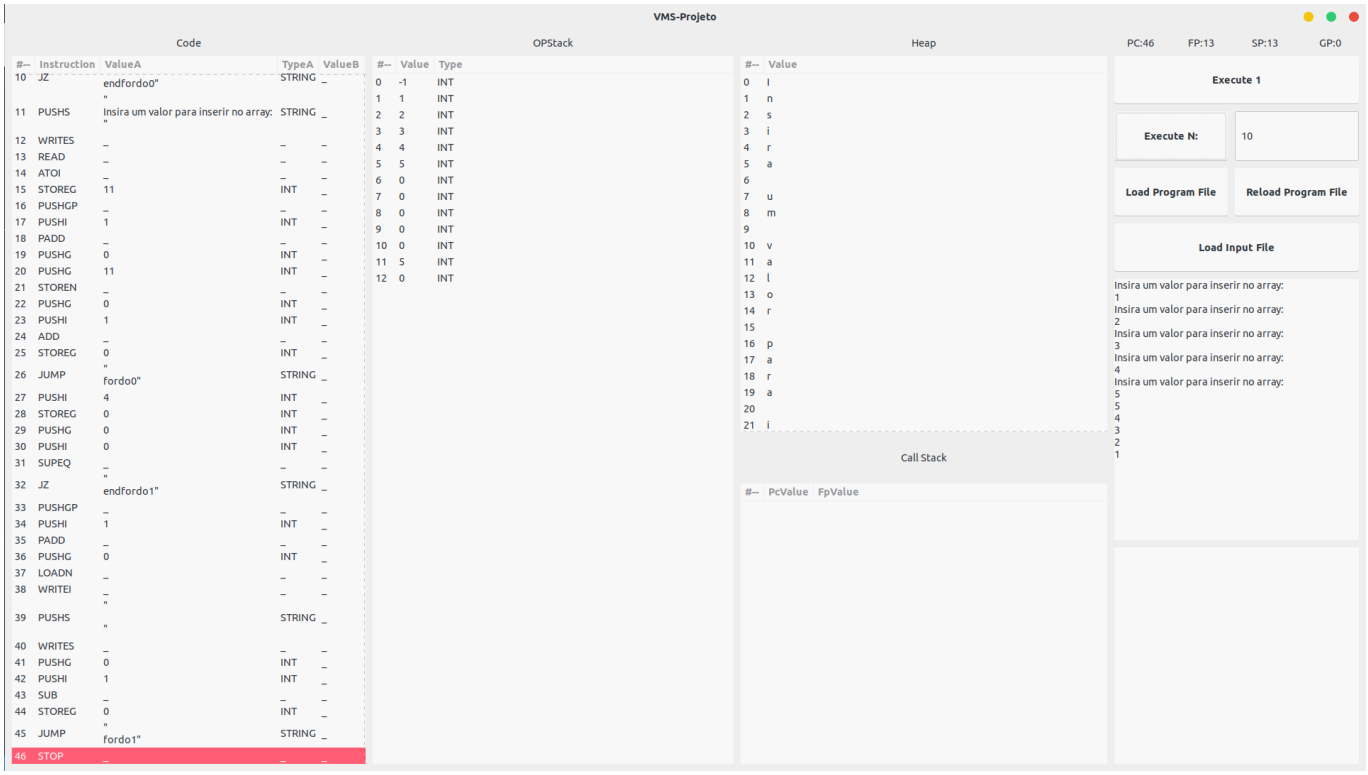

Figura 5.5: Resultado de uma execução da inserção de N números naturais num array e imprimi-lo do fim para o início

5.5 Produtorio

Este programa tem como objetivo ler N (constante do programa) números e calcular e imprimir o seu produtório. Permitiu-nos testar o funcionamento do loop "repeat until".

5.5.1 Código original

```
begindeclares
        int produtorio=1
        int input
        int N=3
enddeclares

repeat
```

```
        read input
        atr produtorio = produtorio * input
        atr N= N - 1
until(N<1)

print produtorio
```

5.5.2 Código VM

```
    pushi 1
    pushi 0
    pushi 3
START
repeat0:
    read
    atoi
    storeg 1
    pushg 0
    pushg 1
    mul
    storeg 0
    pushg 2
    pushi 1
    sub
    storeg 2
    pushg 2
    pushi 1
    inf
    jz repeat0
endrepeat0:
    pushg 0
    writei
STOP
```

5.5.3 Demonstração execução com interface gráfica

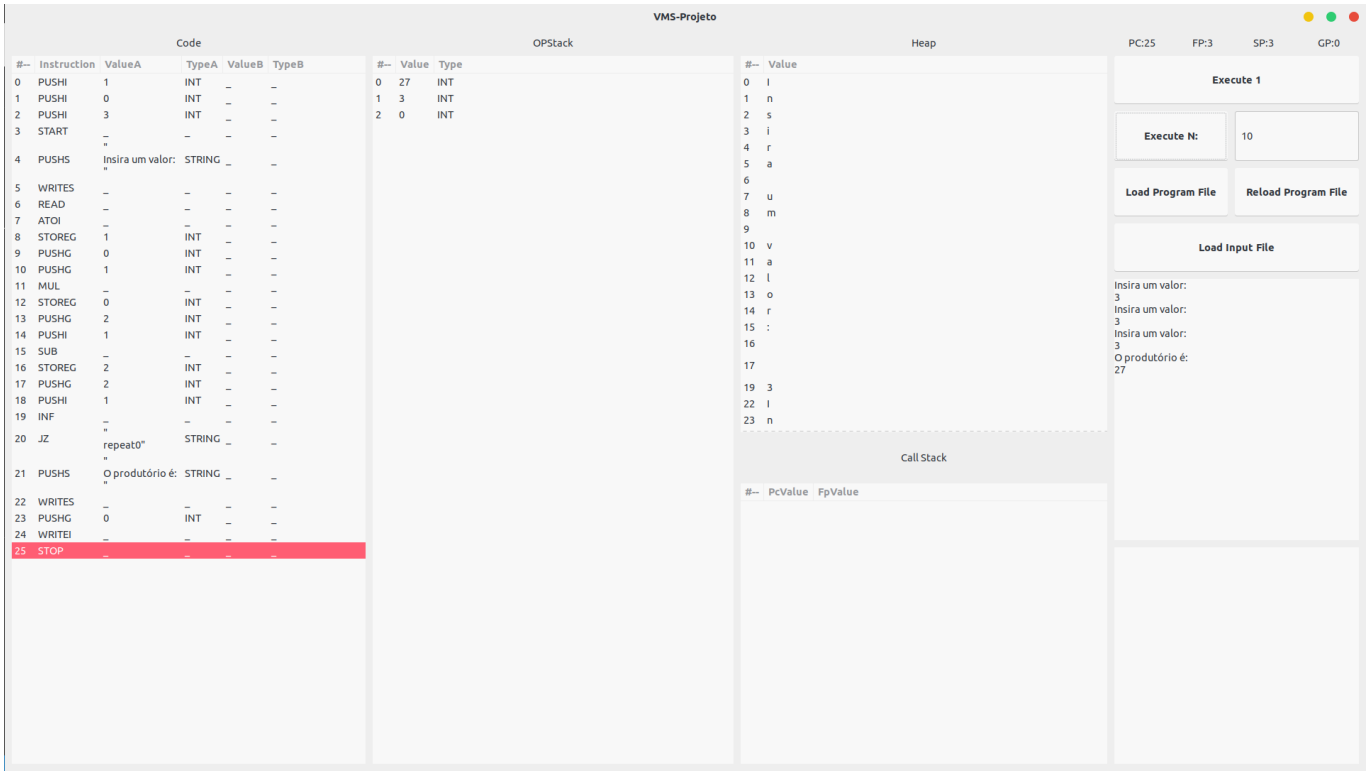

Figura 5.6: Resultado de uma execução de uma sequência de N números naturais e saber o seu produtório

5.6 Programa para demonstrar o nosso compilador

Este programa tem como objetivo demonstrar todas as funcionalidades de que a nossa linguagem e compilador são capazes. Para isso, começa por mostrar um menu ao utilizador com uma lista de funcionalidades (todas as funcionalidades referidas anteriormente) e uma opção para acabar a demonstração. De seguida, o utilizador deve inserir o número correspondente à funcionalidade que quer executar. No fim de cada execução de uma funcionalidade, o programa volta ao menu até que o utilizador selecione a opção para sair. Este programa permitiu-nos verificar, quer o funcionamento de estruturas condicionais aninhadas, quer o funcionamento de ciclos encadeados. É de longe, o programa que melhor demonstra o potencial da nossa linguagem e compilador. O código fonte e VM deste programa foram colocados em apêndice devido à sua grande dimensão

```
Bem vindo à demonstração do compilador do grupo 57!

1 - ler 4 números e dizer se podem ser os lados de um quadrado
2 - ler um inteiro N, depois ler N números e escrever o menor deles
3 - ler N (constante do programa) números e calcular e imprimir o seu produtório.
4 - contar e imprimir os números ímpares de uma sequência de números naturais.
5 - ler e armazenar N números num array; imprimir os valores por ordem inversa.
10 - sair da demonstração.

Insira a sua opção:
```

Figura 5.7: Demonstração do menú do programa de testes

Capítulo 6

Conclusão

Com a finalização deste trabalho prático, o nosso grupo ficou com uma perceção bastante melhor da complexidade que está envolvida na criação de um compilador, neste caso em específico para uma linguagem de programação imperativa. Para além disso, este projeto permitiu-nos aumentar a experiência em engenharia de linguagens e em programação generativa, reforçando assim a nossa capacidade de escrever gramáticas, quer independentes do contexto (GIC), quer tradutoras GT. Outro aspeto importante de se referir é que este trabalho aumentou o nosso conhecimento de 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. Por último, desafiou a nossa capacidade de escrever gramáticas independentes do contexto que satisfaçam a condição LR() usando BNF-puro.

Apêndice A

Lex

```
import ply.lex as lex

reserved = {
    'int': 'INT',
    'print': 'PRINT',
    'read': 'READ',
    'arrayInt': 'ARRAYINT',
    'atr': 'ATR',
    'if': 'IF',
    'else': 'ELSE',
    'endif': 'ENDIF',
    'while': 'WHILE',
    'do': 'DO',
    'endwhile': 'ENDWHILE',
    'repeat': 'REPEAT',
    'until': 'UNTIL',
    'for': 'FOR',
    'endfor': 'ENDFOR',
    'begindeclares': 'BEGINDECLARES',
    'enddeclares': 'ENDDECLARES',
    'and': 'AND',
    'or': 'OR',
    'not': 'NOT',
    'prints': 'PRINTS',

}

tokens = ['VAR', 'NUM', 'STRING', 'SIGNEDNUM'] + list(reserved.values())

literals = ['(', ')', '+', '-', '*', '/', '=',
            ';', '[', ']', '<', '>', '!']


def t_VAR(t):
    r'[a-zA-Z]\w*'
    t.type = reserved.get(t.value, 'VAR')    # Check for reserved words
    return t
```

```
t_NUM = r'\d+'

t_SIGNEDNUM = r'\([+\-]?\d+\)'


t_ignore = " \t\n"


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


lexer = lex.lex()
```

Apêndice B

YACC

```
import ply.yacc as yacc
from compilador_lex import tokens
import sys
import re

stack_position = 0
if_counter = 0
if_stack = []
while_stack = []
while_counter = 0
repeat_stack = []
repeat_counter = 0
for_stack = []
for_counter = 0


def getSignedNum(x):
    res = re.fullmatch(r'\(([+\-]?\d+)\)', x)
    return res.group(1)


def push(list, element):
    list.append(element)


def pop(list):
    return list.pop()


def top(list):
    return list[len(list) - 1]


def p_Programa_Operacoes(p):
    "Programa : Operacoes"
    p[0] = p[1]


def p_Operacoes(p):
```

```
    "Operacoes : Operacoes Operacao"
    p[0] = p[1] + p[2]


def p_Operacoes_empty(p):
    "Operacoes : "
    p[0] = ''


def p_Operacao_BeginDeclares(p):
    "Operacao : BEGINDECLARES"
    p[0] = ''


def p_Operacao_EndDeclares(p):
    "Operacao : ENDDECLARES"
    p[0] = 'START\n'


def p_Operacao_Read(p):
    "Operacao : READ VAR"
    global variables
    atributes = parser.variables[p[2]]
    p[0] = '    read\n' + '    atoi\n' + \
        '    storeg ' + str(atributes[1]) + '\n'


def p_Operacao_ReadToArray(p):
    "Operacao : READ VAR '[' EXP ']'"
    atributes = parser.variables[p[2]]
    p[0] = '    pushgp\n' + '    pushi ' + \
        str(atributes[1]) + '\n' + '    padd\n' + p[4] + \
        '    read\n' + '    atoi\n' + '    storen\n'


def p_Operacao_ReadToBiArray(p):
    "Operacao : READ VAR '[' EXP ']' '[' EXP ']'"
    atributes = parser.variables[p[2]]
    p[0] = '    pushgp\n' + '    pushi ' + str(atributes[1]) + \
    '\n' + '    padd\n' + p[4] + str(
        atributes[3]) + '  mul\n' + str(atributes[3]) + '  add\n' + \
        '    read\n' + '    atoi\n' + '    storen\n'


def p_Operacao_BeginIF(p):
    "Operacao : IF '(' CONDICOES ')'"
    global if_counter, if_stack
    push(if_stack, if_counter)
    p[0] = p[3]+'    jz ' + 'else' + str(if_counter) + '\n'
    if_counter += 1



def p_CONDICOES(p):
```

```
    "CONDICOES : CONDICAO"
    p[0] = p[1]


def p_CONDICOES_NOT(p):
    "CONDICOES : NOT CONDICOES"
    p[0] = p[2] + '    not\n'


def p_CONDICOES_GROUP(p):
    "CONDICOES : '(' CONDICOES ')'"
    p[0] = p[2]


def p_CONDICOES_AND(p):
    "CONDICOES : CONDICOES AND CONDICOES"
    p[0] = p[1] + p[3] + '    mul\n'


def p_CONDICOES_OR(p):
    "CONDICOES : CONDICOES OR CONDICOES"
    p[0] = p[1] + p[3] + '    add\n'


def p_Operacao_Repeat(p):
    "Operacao : REPEAT"
    global repeat_counter
    global repeat_stack
    push(repeat_stack, repeat_counter)
    p[0] = 'repeat' + str(repeat_counter) + ':\n'
    repeat_counter += 1


def p_Operacao_Until(p):
    "Operacao : UNTIL '(' CONDICOES ')'"
    global repeat_stack
    global repeat_counter
    counter = pop(repeat_stack)
    p[0] = p[3] + '    jz ' + 'repeat' + \
        str(counter) + '\n' + 'endrepeat' + str(counter) + ':\n'


def p_Operacao_For(p):
    "Operacao : FOR1 FOR3"
    p[0] = p[1] + p[2]


tmp = ""


def p_FOR3(p):
    "FOR3 : FOR2 Operacao ')' DO"
    global tmp
    tmp = str(p[2])
```

```
    p[0] = p[1]


def p_FOR2(p):
    "FOR2 : CONDICOES ';'"
    global for_counter
    p[0] = p[1] + '    jz endfordo' + str(for_counter) + '\n'
    for_counter = for_counter + 1


def p_Operacao_EndFor(p):
    "Operacao : ENDFOR"
    global for_counter, tmp, for_stack
    counter = pop(for_stack)
    p[0] = tmp + '\n' + '    jump fordo' + \
        str(counter) + '\n' + 'endfordo' + str(counter) + ':\n'


def p_Operacao_While(p):
    "Operacao : WHILEI '(' CONDICOES ')' DO"
    global while_counter
    global while_stack
    push(while_stack, while_counter)
    p[0] = p[1] + p[3] + '    jz endwhile' + str(while_counter) + '\n'
    while_counter = while_counter + 1


def p_WHILEI(p):
    "WHILEI : WHILE"
    global while_counter
    global while_stack
    p[0] = 'whiledo' + str(while_counter) + ':\n'


def p_FOR1_FOR(p):
    "FOR1 : FOR '(' Operacao ';'"
    global for_counter
    global for_stack
    push(for_stack, for_counter)
    p[0] = p[3] + 'fordo' + str(for_counter) + ':\n'


def p_Operacao_EndWhile(p):
    "Operacao : ENDWHILE"
    global while_stack
    global while_counter
    counter = pop(while_stack)
    p[0] = '    jump whiledo' + \
        str(counter) + '\n' + 'endwhile' + str(counter) + ':\n'


def p_CONDICAO_Igual(p):
    "CONDICAO : EXP '=' '=' EXP"
    p[0] = p[1] + p[4] + '    equal\n'
```

```
def p_CONDICAO_Diferente(p):
    "CONDICAO : EXP '!' '=' EXP"
    p[0] = p[1] + p[4] + '    sub\n'


def p_CONDICAO_Menor(p):
    "CONDICAO : EXP '<' EXP"
    p[0] = p[1] + p[3] + '    inf\n'


def p_CONDICAO_Maior(p):
    "CONDICAO : EXP '>' EXP"
    p[0] = p[1] + p[3] + '    sup\n'


def p_CONDICAO_MenorIgual(p):
    "CONDICAO : EXP '<' '=' EXP"
    p[0] = p[1] + p[4] + '    infeq\n'


def p_CONDICAO_MaiorIgual(p):
    "CONDICAO : EXP '>' '=' EXP"
    p[0] = p[1] + p[4] + '    supeq\n'


def p_Operacao_else(p):
    "Operacao : ELSE Operacoes"
    global if_stack
    counter = top(if_stack)
    p[0] = '    jump ' + 'endIf' + \
        str(counter) + '\n' + 'else' + str(counter) + ':\n' + p[2]


def p_Operacao_EndIF(p):
    "Operacao : ENDIF"
    global if_stack
    counter = pop(if_stack)
    p[0] = 'endIf' + str(counter) + ':\n'


def p_Operacao_int_zero(p):
    "Operacao : INT VAR"
    p[0] = '    pushi 0\n'
    global stack_position
    parser.variables.update({p[2]: ['int', stack_position]})
    stack_position = stack_position + 1


def p_Operacao_int(p):
    "Operacao : INT VAR '=' EXP"
    p[0] = p[4]
    global stack_position
```

```
    parser.variables.update({p[2]: ['int', stack_position]})
    stack_position = stack_position + 1


def p_Operacao_arrayInt(p):
    "Operacao : ARRAYINT VAR '[' NUM ']'"

    global stack_position
    parser.variables.update({p[2]: ['arrayInt', stack_position, p[4]]})
    stack_position = stack_position + int(p[4])
    p[0] = '    pushn ' + p[4] + '\n'


def p_Operacao_BiArrayInt(p):
    "Operacao : ARRAYINT VAR '[' NUM ']' '[' NUM ']'"
    global stack_position
    aux = int(p[4]) * int(p[7])
    parser.variables.update(
        {p[2]: ['arrayInt', stack_position, str(aux), p[4]]})
    stack_position = stack_position + aux
    p[0] = '    pushn ' + str(aux) + '\n'


def p_Operacao_Print_EXP(p):
    "Operacao : PRINT EXP"
    p[0] = p[2] + '    writei\n'


def p_Operacao_Atribuir(p):
    "Operacao : ATR VAR '=' EXP"
    atributes = parser.variables[p[2]]
    p[0] = p[4] + '    storeg ' + str(atributes[1]) + '\n'


def p_Operacao_Atribuir_Array(p):
    "Operacao : ATR VAR '[' EXP ']' '=' EXP"
    atributes = parser.variables[p[2]]
    p[0] = '    pushgp\n' + '    pushi ' + \
        str(atributes[1]) + '\n' + '    padd\n' + p[4] + p[7] + '    storen\n'


def p_Operacao_Atribuir_BiArray(p):
    "Operacao : ATR VAR '[' EXP ']' '[' EXP ']' '=' EXP"
    atributes = parser.variables[p[2]]
    p[0] = '    pushgp\n' + '    pushi ' + str(atributes[1]) + '\n' + '    padd\n' + p
   [4] + \
        '    pushi ' + atributes[3] + '\n' + '    mul\n' + \
        p[7] + '    add\n' + p[10] + '    storen\n'


def p_Operacao_PRINTS(p):
    "Operacao : PRINTS STRING "
    p[0] = '    pushs ' + p[2] + '\n    writes\n'
```

```
def p_EXP(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'


def p_TERMO(p):
    "EXP : TERMO"
    p[0] = p[1]


def p_TERMO_mul(p):
    "TERMO : TERMO '*' FACTOR"
    p[0] = p[1] + p[3] + '    mul\n'


def p_TERMO_div(p):
    "TERMO : TERMO '/' FACTOR"
    p[0] = p[1] + p[3] + '    div\n'


def p_FACTOR(p):
    "TERMO : FACTOR"
    p[0] = p[1]


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_SIGNEDNUM(p):
    "FACTOR : SIGNEDNUM"
    p[0] = '    pushi ' + getSignedNum(p[1]) + '\n'


def p_FACTOR_VAR(p):
    "FACTOR : VAR"
    atributes = parser.variables[p[1]]
    p[0] = '    pushg ' + str(atributes[1]) + '\n'


def p_FACTOR_ArrayInt(p):
    "FACTOR : VAR '[' EXP ']'"
```

```
    atributes = parser.variables[p[1]]
    p[0] = '   pushgp\n' + '   pushi ' + \
        str(atributes[1]) + '\n' + '   padd\n' + p[3] + '\n' + '   loadn\n'


def p_FACTOR_BiArrayInt(p):
    "FACTOR : VAR '[' EXP ']' '[' EXP ']'"
    atributes = parser.variables[p[1]]
    p[0] = '   pushgp\n' + '   pushi ' + \
        str(atributes[1]) + '\n' + '   padd\n' + p[3] + \
        '   pushi ' + atributes[3] + '\n' + '   mul\n' + \
        p[6] + '   add\n' + '   loadn\n'


def p_error(p):
    print("Syntax error in input: ", p)


parser = yacc.yacc()

parser.variables = {}


def getName(input):
    res = re.match(r'(\w+)\.vm', input)
    return res.group(1)


def printHelpMenu():
    print('''\n\nPara utilizar o compilador: python   .\\compilador_yacc.py
input.vm \n \nOnde input.vm é o ficheiro input.\n
O código assembly para a máquina VM será criado
no ficheiro input_out.vm no diretório vms/\n\n
Opções adicionais: \n\n -s para o compilador não gerar output\n\n''')


silent = 0

if (sys.argv[1] == '-help'):
    printHelpMenu()
else:
    if len(sys.argv) > 2:
        if sys.argv[2] == '-s':
            silent = 1
    f = open('vms/' + getName(sys.argv[1]) + '_out.vm', 'w')
    with open(sys.argv[1]) as input:
        for linha in input:
            if linha != '\n':
                result = parser.parse(linha)
                f.write(result)
                if result and silent == 0:
                    print('Frase valida: \n\n' + result)
    if silent == 0:
        print(parser.variables)
```

```
f.write('STOP\n')

f.close()
```

Apêndice C

Programa de demonstração

C.1 Código fonte

```
begindeclares
        int opcao=0
        int first
        int input
        int counter=0
        int output=1
        int min
        int n
        int produtorio=1
        int N=3
        int aux
        int total=0
        int i
        arrayInt teste[10]
enddeclares

while(opcao != 10) do
        prints "Bem vindo à demonstração do compilador do grupo 57!\n"
        prints "\n"
        prints "1 - ler 4 números e dizer se podem ser os lados de um quadrado\n"
        prints "2 - ler um inteiro N, depois ler N números e escrever o menor deles\n
    "
        prints "3 - ler N (constante do programa) números e calcular e imprimir o seu
     produtório.\n"
        prints "4 - contar e imprimir os números ímpares de uma sequência de números
    naturais.\n"
        prints "5 - ler e armazenar N números num array; imprimir os valores por
    ordem inversa.\n"
        prints "10 - sair da demonstração.\n"
        prints "\n"
        prints "Insira a sua opção:\n"

        read opcao

        if (opcao==1)
                prints "Insira o valor de um lado\n"
```

```
        read first

        while(counter<3 and output==1) do
                prints "Insira o valor de um lado\n"
                read input
                if(input!=first)
                        atr output=0
                else
                endif
                atr counter= counter + 1
        endwhile

        if(counter == 3 and output==1)
                prints "Podem ser lados de um quadrado\n"
        else
                prints "Não podem ser lados de um quadrado\n"
        endif

else
        if (opcao==2)
                prints "Quantos números vai inserir?\n"
                read n
                prints "Insira um valor\n"
                read min
                atr n= n - 1

                while(n>0) do
                        prints "Insira um valor\n"
                        read input
                        if (input<min)
                                atr min=input
                        else
                        endif
                        atr n = n - 1
                endwhile

                prints "O mínimo é: \n"
                print min
        else
                if(opcao==3)
                        repeat
                                prints "Insira um valor: \n"
                                read input
                                atr produtorio = produtorio * input
                                atr N= N - 1
                        until(N<1)

                        prints "O produtório é: \n"
                        print produtorio
                else
                        if(opcao==4)
                                for(atr N=0;N<4;atr N= N + 1)do
                                        prints "Insira um valor: \n"
                                        read input
```

```
                                        atr aux = input

                                        while (aux > 0) do
                                                atr aux = aux - 2
                                        endwhile

                                        if (aux==(-1))
                                                prints "É ímpar!: \n"
                                                print input
                                                atr total = total + 1
                                        else
                                        endif
                                endfor
                                prints "Número total de números ímpares: \n"
                                print total

                        else
                                if (opcao==5)
                                        for(atr i=0;i<10;atr i= i + 1)do
                                        prints "Insira um valor para
    inserir no array: \n"
                                        read input
                                        atr teste[i]=input
                                endfor

                                for(atr i=9;i>=0;atr i= i - 1)do
                                        print teste[i]
                                endfor
                        else
                        endif
                endif
        endif
endif
endif

endwhile

prints "Volte sempre!\n"
```

C.2 Código VM

```
pushi 0
pushi 0
pushi 0
pushi 0
pushi 1
pushi 0
pushi 0
pushi 1
pushi 3
pushi 0
```

```
    pushi 0
    pushi 0
    pushn 10
START
whiledo0:
    pushg 0
    pushi 10
    sub
    jz endwhile0
    pushs "Bem vindo à demonstração do compilador do grupo 57!\n"
    writes
    pushs "\n"
    writes
    pushs "1 - ler 4 números e dizer se podem ser os lados de um quadrado\n"
    writes
    pushs "2 - ler um inteiro N, depois ler N números e escrever o menor deles\n"
    writes
    pushs "3 - ler N (constante do programa) números e calcular e imprimir o seu
    produtório.\n"
    writes
    pushs "4 - contar e imprimir os números ímpares de uma sequência de números
    naturais.\n"
    writes
    pushs "5 - ler e armazenar N números num array; imprimir os valores por ordem
    inversa.\n"
    writes
    pushs "10 - sair da demonstração.\n"
    writes
    pushs "\n"
    writes
    pushs "Insira a sua opção:\n"
    writes
    read
    atoi
    storeg 0
    pushg 0
    pushi 1
    equal
    jz else0
    pushs "Insira o valor de um lado\n"
    writes
    read
    atoi
    storeg 1
whiledo1:
    pushg 3
    pushi 3
    inf
    pushg 4
    pushi 1
    equal
    mul
    jz endwhile1
    pushs "Insira o valor de um lado\n"
```

```
    writes
    read
    atoi
    storeg 2
    pushg 2
    pushg 1
    sub
    jz else1
    pushi 0
    storeg 4
    jump endIf1
else1:
endIf1:
    pushg 3
    pushi 1
    add
    storeg 3
    jump whiledo1
endwhile1:
    pushg 3
    pushi 3
    equal
    pushg 4
    pushi 1
    equal
    mul
    jz else2
    pushs "Podem ser lados de um quadrado\n"
    writes
    jump endIf2
else2:
    pushs "Não podem ser lados de um quadrado\n"
    writes
endIf2:
    jump endIf0
else0:
    pushg 0
    pushi 2
    equal
    jz else3
    pushs "Quantos números vai inserir?\n"
    writes
    read
    atoi
    storeg 6
    pushs "Insira um valor\n"
    writes
    read
    atoi
    storeg 5
    pushg 6
    pushi 1
    sub
    storeg 6
```

```
whiledo2:
    pushg 6
    pushi 0
    sup
    jz endwhile2
    pushs "Insira um valor\n"
    writes
    read
    atoi
    storeg 2
    pushg 2
    pushg 5
    inf
    jz else4
    pushg 2
    storeg 5
    jump endIf4
else4:
endIf4:
    pushg 6
    pushi 1
    sub
    storeg 6
    jump whiledo2
endwhile2:
    pushs "O mínimo é: \n"
    writes
    pushg 5
    writei
    jump endIf3
else3:
    pushg 0
    pushi 3
    equal
    jz else5
repeat0:
    pushs "Insira um valor: \n"
    writes
    read
    atoi
    storeg 2
    pushg 7
    pushg 2
    mul
    storeg 7
    pushg 8
    pushi 1
    sub
    storeg 8
    pushg 8
    pushi 1
    inf
    jz repeat0
endrepeat0:
```

```
    pushs "O produtório é: \n"
    writes
    pushg 7
    writei
    jump endIf5
else5:
    pushg 0
    pushi 4
    equal
    jz else6
    pushi 0
    storeg 8
fordo0:
    pushg 8
    pushi 4
    inf
    jz endfordo0
    pushs "Insira um valor: \n"
    writes
    read
    atoi
    storeg 2
    pushg 2
    storeg 9
whiledo3:
    pushg 9
    pushi 0
    sup
    jz endwhile3
    pushg 9
    pushi 2
    sub
    storeg 9
    jump whiledo3
endwhile3:
    pushg 9
    pushi -1
    equal
    jz else7
    pushs "É ímpar!: \n"
    writes
    pushg 2
    writei
    pushg 10
    pushi 1
    add
    storeg 10
    jump endIf7
else7:
endIf7:
    pushg 8
    pushi 1
    add
    storeg 8
```

```
    jump fordo0
endfordo0:
    pushs "Número total de números ímpares: \n"
    writes
    pushg 10
    writei
    jump endIf6
else6:
    pushg 0
    pushi 5
    equal
    jz else8
    pushi 0
    storeg 11
fordo1:
    pushg 11
    pushi 10
    inf
    jz endfordo1
    pushs "Insira um valor para inserir no array: \n"
    writes
    read
    atoi
    storeg 2
    pushgp
    pushi 12
    padd
    pushg 11
    pushg 2
    storen
    pushg 11
    pushi 1
    add
    storeg 11

    jump fordo1
endfordo1:
    pushi 9
    storeg 11
fordo2:
    pushg 11
    pushi 0
    supeq
    jz endfordo2
    pushgp
    pushi 12
    padd
    pushg 11

    loadn
    writei
    pushg 11
    pushi 1
```

```
    sub
    storeg 11

    jump fordo2
endfordo2:
    jump endIf8
else8:
endIf8:
endIf6:
endIf5:
endIf3:
endIf0:
    jump whiledo0
endwhile0:
    pushs "Volte sempre!\n"
    writes
STOP
```