Processamento de Linguagens

MIEI (3º ano de Curso)

Trabalho Prático Nº2

(GIC/GT + Compiladores)

Grupo nº70

Ana César (a86038)

Margarida Faria (a71924)

30 de maio de 2021

Resumo

O presente documento pretende apresentar todo o processo de desenvolvimento, bem como a solução atingida, do que foi o segundo trabalho prático da Unidade Curricular de Processamento de Linguagens.
O projeto consiste no desenvolvimento de uma linguagem baseada no paradigma imperativo da programção, bem como a construção de uma gramática independente do contexto e um compilador que gere código com instruções em *Assembly*, utilizando para isso a linguagem desenvolvida pelo grupo e a linguagem *Python*.
Foi comtemplado um enunciado sobre o qual foram apresentados o problema e os objetivos a considerar na implementação da solução. Sobre o problema qual foram retirados os requisitos fundamentais para a construção da solução.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

O presente documento pretende apresentar o segundo trabalho prático da Unidade Curricular de Processamento de Linguagens.
Para este projeto foi necessário implementar uma linguagem imperativa simples, desenvolvida pelo grupo, e utilizar a linguagem *Python*, mais precisamente a versão *PLY* que este contém, para aceder aos módulos *Lex* e *Yacc*, oara implementar uma gramática independente de contexto e um compilador que gera código Assembly, que irá ser passado a uma máquina virtual fornecida pelos docentes.

1.1 Enquadramento

Um *compilador* ou *interpretador* para uma linguagem de programação pode ser decomposto em duas partes: ler o programa base e descobrir a sua estrutura; processar essa estrutura, para gerar um programa-alvo.
O **Lex** e o **Yacc** são ferramentas utilizadas capazes de gerar programas para resolver a primeira parte, uma vez que o *Lex* é utilizado para dividir o ficheiro base em *tokens*, enquanto que o *Yacc* encontra a hierarquia estrutural do programa.
O **PLY** é uma implementação em *Python* das ferramentas *Lex* e *Yacc*, usadas atualmente para desenvolver **parsers** e **compiladores**. O Algoritmo de parsing é baseado no algoritmo **LALR (Look-Ahead LR)** usado por várias ferramentas de *Yacc*.

1.2 Contexto

No âmbito da Unidade Curricular Processamento de Linguagens, foi proposto o trabalho prático que apresentamos neste documento sobre *(GIC/GT + Compiladores)*.
O contexto deste projeto está presente nas áreas *engenharia de linguagens* e *programação generativa (gramatical*, no sentido de desenvolver um processador de linguagens a partir de uma gramática tradutora, mais precisamente implementar um **compilador** capaz de gerar código *Assembly* para uma máquina virtual fornecida pelos docentes.

1.3 Objetivos

Os objetivos concretos do projecto são os seguintes:

- Aumentar a capacidade do grupo em escrever gramáticas independentes de contexto, e gramáticas tradutoras;
- Seguir o método de *tradução dirigida pela sintaxe* para desenvolver um processador de linguagens;
- Desenvolver um compilador gerando código para uma máquina de stack virtual;
- Utilizar geradores de compiladores da versão PLY do Python, de nome Lex e Yacc;
- Aumentar a capacidade de escrever gramáticas independentes de contexto, que satisfaçam a condição LR() usando BNF(Backus-Naur Form) puro, que se entende por um modo formal de descrever linguagens formais;
- Escrever corretamente a documentação para o projecto, na linguagem *Latex*.

1.4 Problema

Pretende-se que seja definida uma linguagem de programação imperativa simples, ao gosto do nosso grupo, tenho em consideração alguns fatores que esta linguagem deverá satisfazer, sendo eles os seguintes:

- A possibilidade de declarar variáveis atómicas do tipo inteiro, sobre os quais podemos realizar operações aritméticas, relacionais e lógicas;
- Efetuar instruções algorítmicas tais como a atribuição do valor de expressões numéricas a variáveis;
- Ler do standart input e escrever no standart output;
- Efetuar instruções condicionais para controlo do fluxo de execução;
- Efetuar instruções cíclicas para controlo do fluxo de execução, permitinhdo o seu aninhamento.
- Adicionalmente, o grupo optou pela seguinte funcionalidade de duas possiveís à escolha:
  - Declarar e manusear variáveis estruturadas do tipo array (1 ou 2 dimensões) de inteiros, aos quais é apenas permitida a operação de indexação.

Realçamos que a escolha considerada pelo grupo terá impacto no decorrer do projeto, que iremos abordar posteriormente no documento, na secção de Análise e Especificação.

1.5 Resultados

Face ao problema proposto neste projecto, o grupo procedeu a uma análise, que começou por identificar os requisitos, idealizamos a primeira solução.
Seguidamente, definimos a nossa linguagem, baseada no paradigma de programação imperativo, e procedemos à concepção de uma gramática, bem como o analisador léxico e compilador.
Posteriormente neste documento vão ser expostas estas fases, bem como a conceção final da solução.

1.6 Estrutura do Relatório

Na fase inicial do documento, alusivo ao capítulo 1, é feita a introdução do problema, no qual é apresentada o enquadramento do mesmo, o seu contexto e ainda uma explicação formal sobre o mesmo. Nesta fase é ainda exposto o objetivo do relatório, bem como os resultados atingidos pelo grupo e ainda o presente tópico, referente a sua estruturação.
No capítulo 2, é feita uma análise detalhada do problema em questão, onde são especificados os requisitos do mesmo.
No capítulo 3, apresentamos a gramática independente de contexto, o analisador léxico e a sua hierarquia estrutural, que constituem o compilador, desenvolvidos para a determinação do problema.
De seguida, o capítulo 4 procura relacionar o código previamente implementado, com as tomadas de decisão que o grupo teve ao longo da procura da resposta aos obstáculos encontrados, em todo o projecto, bem como os testes feitos e os resultados respetivos.
No capítulo 5 termina o documento, com um pequeno resumo de tudo o que foi abordado, conclusões e trabalho futuro.

Capítulo 2

Análise e Especificação

Neste capítulo vamos abordar em detalhe o problema que nos foi proposto, e apresentar o levantamento de requisitos feito pelo grupo face ao problema em questão.

2.1 Descrição informal do problema

Pretendemos desenvolver um processador de linguagens capaz de gerar código na linguagem *Assembly*, utilizando a máquina virtual fornecida pelos docentes.
Como tal, recorrendo às ferramentas previamente descritas, *Lex* e *Yacc*, desenvolvemos uma gramática independente do contexto, bem como uma linguagem criada pelo grupo, de modo a cumprir os objetivos do projecto.

2.2 Especificação dos Requisitos

Face as suas exigências impostas, apresentamos os seguintes requisitos funcionais do programa, recolhidos através do enunciado do projeto.

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 impares de uma sequência de números naturais.
5. O próximo requisito resulta da escolha previamente explicitada neste documento, referente à capacidade de declarar e manipular estuturas do tipo array de inteiros, e é o seguinte:
    - Ler e armazenar N números num array; imprimir os valores por ordem inversa.

Capítulo 3

Concepção/desenho da Resolução

Neste capítulo vamos apresentar de modo específico, o processo e os procedimentos inerentes à implementação da solução, sobre a qual foi desenvolvida uma gramática independente de contexto, e onde foram implementados o *Lex* e o *Yacc* presentes na versão do *PLY*, na linguagem *Python*, tirando partido dos módulos **'lex','yacc'** e **'sys'**.

3.1 Linguagem Desenvolvida

Foi um requisito do projeto o desenvolvimento, de forma simples, de uma linguagem baseada no paradigma de programação imperativa, e foi dada ao grupo total liberdade nesta execução.
Deste modo, o grupo inspirou-se na linguagem imperativa **C**, mas com um *twist* próprio e singular.
As **declarações** de variáveis efetuam-se do seguinte modo:

- se for uma variável do tipo inteiro : **int a;**
- se a variável for um array de inteiros : **int a[SIZE];**

É possível a enumeração de variáveis do mesmo tipo, e como estamos perante inteiros, a seguinte declaração é válida: **int a,b, c[SIZE];**.

Para efetuar uma **atribuição**, tem de ser feita no delimitar do que representa o *Body* do programa, e é efetuada a seguinte instrução : **a = 10;**

Relativamente à leitura do **stdin**, e escritas no **stdout**, as instruções que as representam são, respectivamente, as seguintes : **read(a);** e **print(a);**. Outra instrução possivel seria atribuir um inteiro : **read(10);** e **print(10);**.

A implementação de uma expressão **condicional** é feita da seguinte forma: **if(Condição){Instrução;}** ou ainda **if(Condição){Instrução;}else{OutraInstrução;}**.

Para representar uma expressão **cíclica**, o algoritmo fornecido pelos docentes fez com que implementassemos o ciclo **Repeat...Until**, e procedemos do seguinte modo: **repeat{Instrução;}until(Condição)**.
Realçamos que o aninhamento de instruções é válido, sendo possíveis múltiplas execuções e responder aos requisitos que foram apresentados previamente.

3.2 GIC

Seguindo como referência uma gramática do tipo GIC, o passo inicial foi definir o conjunto de símbolos que constituem a gramática, que se dividem em duas categorias, símbolos terminais e não terminais. Assim, é apresentada de seguinda a gramática desenvolvida no sentido de auxiliar à construção da solução:

```
Language -> Declarations Functionality
          ;


Declarations -> STARTDECL BodyDecls ENDDECL
               ;

Functionality -> STARTBODY Instructions ENDBODY
               ;


BodyDecls -> BodyDecls BodyDecl
            | empty
            ;

BodyDecl -> INT DeclDef TERMINATOR
            ;

DeclDef -> DeclIds Enumerate
          ;

DeclIds -> ID
         | ID '[' NUM ']'
         ;

Enumerate -> ',' DeclDef
            | empty
            ;

Instructions -> Instructions Instruction
              | empty
              ;

Instruction -> BodyDecl
              | INT ID '[' ID ']' TERMINATOR
              | Atr TERMINATOR
              | Repeat
              | If
              | Print TERMINATOR
              | Read TERMINATOR
              ;

Atr -> Id '=' Exp
      | Array '=' Exp
      ;

If -> IF '(' Cond ')' '{' Instructions '}' Else
```

```
     ;

Else -> ELSE '{' Instructions '}'"
        | empty
        ;

Repeat -> REPEAT '{' Instructions '}' UNTIL '(' Cond ')'
         ;

Print -> PRINT '(' IdNum ')'
        | PRINT '(' Array ')'
        ;

Read -> READ '(' ID ')'
       | READ '(' Array ')'
       ;

Cond -> Cond OR Cond2
      | Cond2
      ;


Cond2 -> Cond2 AND Cond3
      | Cond3
      ;

Cond3 -> NOT Cond
        | ExpRelacional
        | '(' Cond ')'
        ;

ExpRelacional -> Exp '>'  Exp
                | Exp '<' Exp
                | Exp '<' '=' Exp
                | Exp '>' '=' Exp
                | Exp '!' '=' Exp
                | Exp '=' '=' Exp
                | Exp
                ;


Exp   -> Exp '+' Termo
       | Exp '-' Termo
       | Termo
       ;

Termo -> Termo '*' Fator
       | Termo '\%' Fator
       | Fator
       ;

Fator -> '(' Exp ')'
        | IdNum
        | Array
```

```
        ;


IdNum -> Num
       | Id
       ;

Num -> NUM
      ;

Id -> ID
     ;

Array -> ArrayNum
        | ArrayId
        ;

ArrayNum -> ID '[' NUM ']'
           ;

ArrayId -> ID '[' ID ']'
          ;
```

3.3 Analisador Léxico

3.3.1 Símbolos Literais

Os símbolos que representam o conjunto de literais são os seguintes:

```
['(', ')' , ',' , '=' , '>' , '<' , '+' , '*', '-' , '\%' , '{' , '}' , '!' , '[' , ']', ';']
```

Os símbolos não terminais aqui apresentados representam todos aqueles necessários para a funcionalidade da linguagem implementada pelo grupo, e ainda os símbolos $<$, $>$, $=$ e ! para permitir a representação de expressões lógicas.

3.3.2 Tokens

Os *tokens* definidos neste projeto são os símbolos que apresentamos de seguida:

```
tokens = ['TERMINATOR','ID','NUM']
```

3.3.3 Palavras Reservadas

As palavras que consituem o tipo de palavra reservada são as seguintes:

```
reservadas = {"int" : "INT",
              "STARTDECL" : "STARTDECL",
              "ENDDECL" : "ENDDECL",
              "STARTBODY" : "STARTBODY",
```

```
"ENDBODY" : "ENDBODY",
"or" : "OR",
"and" : "AND",
"not" : "NOT",
"if" : "IF",
"else" : "ELSE",
"repeat" : "REPEAT",
"until" : "UNTIL",
"print" : "PRINT",
"read" : "READ"
}]
```

3.3.4 Expressões Regulares

As expressões regulares definidas no contexto do analisador léxico dizem respeito aos símbolos terminais desenvolvidos e apresentados previamente, e aos quais pertencem a esse grupo os tokens e as palavras reservadas:

```
t_STARTDECL = r'STARTDECL'
t_ENDDECL = r'ENDDECL'
t_STARTBODY = r'STARTBODY'
t_ENDBODY = r'ENDBODY'
t_INT = r'int'
t_TERMINATOR = r';'
t_OR = r'or'
t_AND = r'and'
t_NOT = r'not'
t_NUM = r'\d+'
t_IF = r'if'
t_REPEAT = r'repeat'
t_UNTIL = r'until'
t_PRINT = r'print'
t_READ = r'read'
t_ignore = " \n\t\r"
```

As expressões regulares correpondentes ao token reservado {*STARTDECL* e *ENDDECL*} pretendem delimitar o espaço de declarações de variáveis do tipo inteiro e do tipo array de inteiros.
As expressões referentes a {*STARTBODY* e *ENDBODY*}, por sua vez, delimitam o corpo do programa, onde podem ser efetuadas instruções, tais como {*READ*, *PRINT*}, condicionais tal como o {*IF*} e cíclicas como {*REPEAT...UNTIL*}.
Para representar expressões foram desenvolvidas os símbolos {*AND*, *OR* e *NOT*} que representam as expressões relacionais, bem como os símbolos {+, -, *, /} que representam expressões aritméticas. O símbolo *INT* pretende especificar o número inteiro, enquanto o símbolo *NUM* representa todos os números, e ainda o *TERMINATOR* que representa o símbolo *;*.
Como passíveis de serem ignorados, integram os seguintes símbolos: {\t\r\n}.
Foi ainda implementada as seguintes funções responsáveis por criar *tokens* com uma ação especial atribuída:

```
def t_ID(t):
    r'[a-zA-Z][a-zA-Z0-9]*'
    if t.value in reservadas:
        t.type = reservadas [t.value ]
```

```
    return t

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

3.3.5 Estruturas de Dados

Foi implementado um dicionário, presente no módulo *Lex*, e referente às palavras reservadas, expostas previamente nesta secção. Outra estrutura de Dados deste módulo são *arrays*, implementados com o objetivo de armazenar os *tokens* e os *literais*.

3.4 Compilador

O compilador é composto pela análise léxica, sintática e semântica.
A análise léxica e sintática são responsáveis por analisar uma sequência de entrada, para determinar a sua estrutura gramatical. Esta análise está pendente de uma gramática e da verificação correta dos erros semânticos.

3.4.1 Especificação do Compilador

Através da ferramenta *YACC*, foram implementadas produções capazes de responder a todos os requisitos que o programa deve obter respostas.

Início e Término de Geração do Programa

A delimitação do nosso programa é dividida em duas fases distintas: a parte onde é feita a declaração de variáveis, *Declarations*, do tipo inteiro, ou do tipo array de inteiros com um tamanho fixo, e a parte referente à *Functionality* engloba as atribuições às variáveis que foram declaradas previamente, bem como a manipulação de variáveis para obter resultados.

```
Language -> Declarations Functionality
          ;

Declarations -> STARTDECL BodyDecls ENDDECL
                ;

Functionality -> STARTBODY Instructions ENDBODY
                ;
```

Declaração de Variáveis

Na parte reservada à declaração de variáveis, foi desenvolvida uma estrutura chamada *BodyDecls* que permite ter uma ou várias declarações de variáveis do tipo inteiro e do tipo array de inteiros.
É possibilitado pela nossa linguagem fazer uma lista de enumeração de variáveis, do tipo *int a,b,c;*, ou ainda *int a[10],b,c;* .

```
BodyDecls -> BodyDecls BodyDecl
```

```
            | empty
            ;

BodyDecl -> INT DeclIds TERMINATOR
            ;

Def -> DeclIds Enumerate
       ;

DeclIds -> ID
         | ID '[' NUM ']'

Enumerate -> ',' DeclDef
            | empty
            ;
```

Atribuições

As atribuições devem ser feitas seguido das declarações iniciais, acima do Corpo do Programa.

```
Atr -> Id '=' Exp
      | Array '=' Exp
      ;

Id -> ID
    ;

Array -> ArrayNum
        | ArrayId

ArrayNum -> ID '[' NUM ']'
          ;

ArrayId -> ID '[' ID ']'
          ;
```

Leitura e Escrita

Para este requisito foram construidas as seguintes estruturas gramaticais:

```
    Print -> PRINT '(' IdNum ')'
            | PRINT '(' Array ')'
            ;

    Read -> READ '(' ID ')'
        | READ '(' Array ')'
        ;
```

Dependentes de uma Condição:

A forma como foram definidas as condições, usadas quer numa instrução condicional ou cíclica, foi igual, e corresponde à seguinte parte da gramática: É de salientar que foram desenvolvidos as produções *ExpRelacional* para responder a expressões relacionais que o programa deve reconhecer, bem como as produções *Exp* que contêm expressões aritméticas, que o programa deve igualmente reconhecer.

```
Cond -> Cond OR Cond2
    | Cond2


Cond2 -> Cond2 AND Cond3
    | Cond3

Cond3 -> NOT Cond
        | ExpRelacional
        | '(' Cond ')'

ExpRelacional -> Exp '>'  Exp
                | Exp '<' Exp
                | Exp '<' '=' Exp
                | Exp '>' '=' Exp
                | Exp '!' '=' Exp
                | Exp '=' '=' Exp
                | Exp


Exp   -> Exp '+' Termo
    | Exp '-' Termo
    | Termo

Termo -> Termo '*' Fator
        | Termo '\%' Fator
        | Fator

Fator -> '(' Exp ')'
         | IdNum
         | Array


IdNum -> Num
    | Id
```

Condicionais

A expressão condicional representada no nosso programa tem o comportamento da instrução *if* da *linguagem C*, sendo que é permitido existir um if para verificar se uma condição se realiza e efetuar instruções caso a condição se verificar. É ainda permitido efetuar um *if(cond) instrução; else instrução;*.

```
If -> IF '(' Cond ')' '{' Instructions '}' Else
        ;

Else -> ELSE '{' Instructions '}'"
```

```
        | empty
        ;
```

Cíclicas

O ciclo funciona de forma similar à condição. A estrutura é composta por um *Repeat*, seguido de um conjunto de operações ou instruções, efetuadas enquanto a condição do *Until* se verificar.

```
Repeat -> REPEAT '{' Instructions '}' UNTIL '(' Cond ')'
         ;
```

3.4.2 Estruturas de Dados

Tirando proveito da linguagem *Python*, implementamos um dicionário denonimado *table*, com o objetivo de armazenar o valor do *offset*, valor este necessário para a computação realizada na máquina virtual.
inteiros do offse Para que isto seja possível, foi necessário criar uma variável para atribuir corretamente este offset, e ainda uma variável *nr*, usada para fazer uma correta atribuição de um número à etiqueta presente em cada instrução referente a ciclos.
Todas estas estruturas e variáveis estão incluídas no *parser*, e é de realçar que estas variáveis estão internas ao nosso parser.

3.5 Funcionamento do programa

Tal como referido anteriormente, o compilador a ser desenvolvido deve ser capaz de gerar código Assembly numa máquina virtual.
Deste modo, é gerado um ficheiro com código a ser introduzido na máquina virtual, onde é processado instrução a instrução no sentido de obter os resultados esperados para cada requisito, através do seguinte comando:

```
python3.9 proj_yacc.py < test.txt > test2.txt
```

O comando apresentado lê o ficheiro test.txt, e o resultado do mesmo é redireccionado para o ficheiro test2.txt.

Capítulo 4

Codificação e Testes

Neste capítulo vamos descrever todas as tomadas de decisão que o grupo tomou na conceção deste trabalho prático, e ainda apresentar imagens dos resultados que obtivemos no programa desenvolvido:

4.1 Abordagens na exibição e respectivos resultados

Nesta secção iremos finalmente, mostrar todos os resultados obtidos e ainda a abordagem de implementação do código fonte na linguagem desenvolvida pelo grupo, para os diferentes programas propostos.

4.1.1 Leitura de 4 números e possível quadrado

O primeiro programa proposto pela equipa docente, trata-se de um programa capaz de ler do *stdin* e armazenar 4 números inteiros e verificar a possibilidade de representarem os lados de um quadrado. Para isto, seguimos uma simples abordagem de verificar a igualdade entre os primeiros três elementos recorrendo ao operador lógico *and* e em caso afirmativo, é feita uma última comparação com o último elemento fornecido.

```
STARTDECL
    int a,b,c,d,res;
ENDDECL
STARTBODY
    read(a);
    read(b);
    read(c);
    read(d);
    res = 0;
    if(a==b and b==c){
        if(c==d){
           res = 1;
        }
    }
    print(res);
ENDBODY
```

Em seguida iremos mostrar o resultado obtido através da impressão para um ficheiro, com auxílio no redirecionamento para ficheiro.

```
pushi 0                 pushg 0
pushi 0                 pushg 1
pushi 0         read    equal
pushi 0         atoi    pushg 1         jump endif1
pushi 0         storeg 2        pushg 2         endif1:
                                equal           jump endif2
start           read    mul             endif2:
read            atoi    jz endif2       pushg 4
atoi            storeg 3        pushg 2         writei
storeg 0                pushg 3
                pushi 0         equal           stop
read            storeg 4        jz endif1
atoi                            pushi 1
storeg 1                        storeg 4
```

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

Este programa deve ser capaz de ler do input números de modo a determinar qual é o mínimo dos números inseridos, sendo o procedimento o seguinte:

```
STARTDECL
int n, num, min;
ENDDECL
STARTBODY
read(n);
read(min);
repeat{
    read(num);
    if(min > num){
        min = num;
    }
    n = n - 1;
}until(n==1)
print (min);
ENDBODY
```

Em seguida iremos mostrar o resultado obtido através da impressão para um ficheiro, com auxílio no redirecionamento para ficheiro.

```
pushi 0         ciclo2:
pushi 0         read                endif1:
pushi 0         atoi                pushg 0
                storeg 1            pushi 1
start                               sub         pushg 2
read            pushg 2             storeg 0    writei
atoi            pushg 1
storeg 0        sup                 pushg 0     stop
                jz endif1           pushi 1
read            pushg 1             equal
atoi            storeg 2            jz ciclo2
storeg 2        jump endif1
```

4.1.3 Ler N (constante do programa) números e calcular e imprimir o seu produtório.

Para este programa, foi feita uma instrução cíclica que permitisse ler números do input, e calcular o seu produtório, sendo que quando o número 1 é lido no input, a execução do programa é terminado e o resultado apresentado, pelo que a abordagem foi a seguinte:

```
STARTDECL
int prod,n;
ENDDECL
STARTBODY
prod = 1;
repeat{
    read(n);
    prod = prod * n;
}until(n==1)
print (prod);
ENDBODY
```

Em seguida iremos mostrar o resultado obtido através da impressão para um ficheiro, com auxílio no redirecionamento para ficheiro.

```
                ciclo1:     pushg 1
                read        pushi 1
pushi 0         atoi        equal
pushi 0         storeg 1    jz ciclo1

start
pushi 1         pushg 0     pushg 0
storeg 0        pushg 1     writei
                mul
                storeg 0    stop
```

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

Para este programa, foram lidos números do input num ciclo, até ser lido o número 0, e foi utilizado o algorimo do módulo da divisão de um número inserido por 2, e se o resultado for 1, esse número é ímpar. Todos os números inseridos são contados, e o resultado desse contador é enviado para o output.
O código desenvolvido foi o seguinte:

```
STARTDECL
int n,num, cont;
ENDDECL
STARTBODY
cont = 0;
read(n);
repeat{
    read(num);
    if(num % 2 == 1){
        cont = cont + 1;
        print(num);
    }
    n = n - 1;
}until(n==0)
print (cont);
ENDBODY
```

Em seguida iremos mostrar o resultado obtido através da impressão para um ficheiro, com auxílio no redirecionamento para ficheiro.

```
pushi 0
pushi 0
pushi 0

start
pushi 0
storeg 2

read
atoi
storeg 0
ciclo2:
read
atoi
storeg 1

pushg 1
pushi 2
mod
pushi 1
equal
jz endif1
pushg 2
pushi 1
add
storeg 2

pushg 1
writei
jump endif1

endif1:
pushg 0
pushi 1
sub
storeg 0
pushg 0
pushi 0
equal
jz ciclo2

pushg 2
writei

stop
```

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

Este programa, de igual modo, lê números do input de maneira cíclica, e os números são armazenados numa variável do tipo array de inteiros. O objetivo é mostrar esse array por ordem inversa dos números que foram innseridos, e o procedimento tomado foi o seguinte:

```
STARTDECL
int n,num,cont,a[10];
ENDDECL
STARTBODY
read(n);
cont = 0;
repeat{
    read(num);
    a[cont] = num;
    cont = cont + 1;
    n = n - 1;
}until(n>0)
cont = cont - 1;
repeat{
    print(a[cont]);
    cont = cont - 1;
}until(cont<0)
ENDBODY
```

Em seguida iremos mostrar o resultado obtido através da impressão para um ficheiro, com auxílio no redirecionamento para ficheiro.

```
pushi 0
pushi 0
pushi 0
pushn 4

start
read
atoi
storeg 0

pushi 0
storeg 2

ciclo1:
read
atoi
storeg 1

pushgp
pushi 3
padd
pushg 2
pushg 1
storen

pushg 2
pushi 1
add
storeg 2

pushg 0
pushi 1
sub
storeg 0

pushg 0
pushi 0
equal
jz ciclo1

pushg 2
pushi 1
sub
storeg 2

ciclo2:
pushgp
pushi 3
padd
pushg 2
loadn
writei

pushg 2
pushi 1
sub
storeg 2

pushg 2
pushi 0
inf
jz ciclo2

stop
```

Capítulo 5

Conclusão

A realização deste projeto permitiu ao grupo tomar consciência das implicações que a área de processamento de linguagens possui, e ainda a compreensão que o desenvolvimento de um linguagem baseada no paradigma imperativo, e do respetivo compilador são tarefas árduas e que requerem um bom planeamento, para atingir os resultados esperados.
A utilização e o conhecimento adquirido sobre as ferramentas presentes no *PLY*, o *Lex* e o *Yacc*, revelaram-se essenciais no desenvolvimento do projeto, sendo que o *Lex* permitiu processar texto, através de expressões regulares, de forma intuitiva e eficiente. Relativamente ao *Yacc*, foi utilizado para desenvolver as produções necessárias para hierarquia e a estruturação do programa.
Deste modo, o grupo sente-se satifeito quer com os resultados obtidos na realização da solução, quer relativamente ao conhecimento obtido na unidade curricular, que se revelou valioso na elaboração deste trabalho. Dito isto, o grupo sente que poderia ter melhorado a implementação da linguagem criada, pois apesar de responder a todos os requisitos, revelou-se uma linguagem bastante básica.

Apêndice A

Apresentação dos Resultados

Código relativo ao ficheiro *Lex*

```
import sys
import ply.lex as lex

tokens = ['TERMINATOR','ID','NUM']

reservadas = {"int" : "INT",
              "STARTDECL" : "STARTDECL",
              "ENDDECL" : "ENDDECL",
              "STARTBODY" : "STARTBODY",
              "ENDBODY" : "ENDBODY",
              "or" : "OR",
              "and" : "AND",
              "not" : "NOT",
              "if" : "IF",
              "else" : "ELSE",
              "repeat" : "REPEAT",
              "until" : "UNTIL",
              "print" : "PRINT",
              "read" : "READ"
              }
tokens = tokens + list(reservadas.values())

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

t_STARTDECL = r'STARTDECL'
t_ENDDECL = r'ENDDECL'
t_STARTBODY = r'STARTBODY'
t_ENDBODY = r'ENDBODY'
t_INT = r'int'
t_TERMINATOR = r';'
t_OR = r'or'
t_AND = r'and'
t_NOT = r'not'
t_NUM = r'\d+'
t_IF = r'if'
t_ELSE = r'else'
t_REPEAT = r'repeat'
```

```
t_UNTIL = r'until'
t_PRINT = r'print'
t_READ = r'read'
t_ignore = " \n\t\r"


def t_ID(t):
    r'[a-zA-Z][a-zA-Z0-9]*'
    if t.value in reservadas:
        t.type = reservadas [t.value ]
    return t

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

#Build the lexer
lexer = lex.lex()

# Reading input
'''
for linha in sys.stdin:
    lexer.input(linha)
    for tok in lexer:
        print(tok)
'''
```

Código relativo ao ficheiro *Yacc*

```
import ply.yacc as yacc
import re

from lex_proj import tokens

def p_Language(p):
    "Language : Declarations Functionality"
    p[0] = p[1] + p[2]


def p_Declarations(p):
    "Declarations : STARTDECL BodyDecls ENDDECL"
    p[0] = p[2]


def p_Functionality(p):
    "Functionality : STARTBODY Instructions ENDBODY"
    p[0] = "\nstart\n" + p[2] + "stop\n"



def p_Body_Decls(p):
    "BodyDecls : BodyDecls BodyDecl"
    p[0] = p[1] + p[2]

def p_Body_Decls_Body_Decl(p):
    "BodyDecls : "
    p[0] = ""

def p_Body_Decl_INT(p):
    "BodyDecl : INT DeclDef TERMINATOR"
    p[0] = p[2]


def p_DeclDef(p):
    "DeclDef : DeclIds Enumerate"
    p[0] = p[1] + p[2]

def p_DeclIds_ID(p):
    "DeclIds : ID"
    p.parser.table[p[1]] = ("int", p.parser.offset, 1, 0)
    p.parser.offset+=1
    p[0] = "pushi 0\n"

def p_DeclIds_Array(p):
    "DeclIds : ID '[' NUM ']'"
    size = int(p[3])
    p.parser.table[p[1]] = ("intArray", p.parser.offset, size, list())
    p.parser.offset+=size
    p[0] = "pushn " + p[3] + "\n"

def p_Enumerate(p):
```

```
    "Enumerate : ',' DeclDef"
    p[0] = p[2]


def p_Enumerate_Empty(p):
    "Enumerate : "
    p[0] = ""

def p_Instructions(p):
    "Instructions : Instructions Instruction"
    p[0] = p[1] + p[2]

def p_Instructions_Instruction(p):
    "Instructions : """
    p[0] = ""

def p_Instruction_Decl(p):
    "Instruction : BodyDecl "
    p[0] = p[1] + "\n"

def p_Instruction_Atr(p):
    "Instruction : Atr TERMINATOR"
    p[0] = p[1] + "\n"


def p_Atr_IdNum(p):
    "Atr : Id '=' Exp"
    p[0] = p[3] + "storeg " + str(sum(map(int, re.findall('\d+', p[1])))) + "\n"


def p_Atr_ArrayNUM(p):
    "Atr : Array '=' Exp"
    p[0] = p[1] + p[3] + "storen\n"


def p_Instruction_Repeat(p):
    "Instruction : Repeat "
    p[0] = p[1] + "\n"


def p_Instruction_If(p):
    "Instruction : If"
    p[0] = p[1] + "\n"

def p_Instruction_Print(p):
    "Instruction : Print TERMINATOR"
    p[0] = p[1] + "\n"

def p_Instruction_Read(p):
    "Instruction : Read TERMINATOR"
    p[0] = p[1] + "\n"


def p_If(p):
```

```
    "If : IF '(' Cond ')' '{' Instructions '}' Else"
    endif = "endif" + str(p.parser.nr)
    else_ = p[8].split(':')[0]
    if(else_ == "") : else_ = endif
    jump = "jump " + endif + "\n"
    p[0] = p[3] + "jz " + else_ + "\n" + p[6] + jump + p[8] + endif + ":"
    p.parser.nr += 1


def p_Else(p):
    "Else : ELSE '{' Instructions '}'"
    else_ = "else" + str(p.parser.nr)
    p[0] = else_ + ":\n" + p[3] + "\n"

def p_ElseEmpty(p):
    "Else : "
    p[0] = ""

def p_Repeat(p):
    "Repeat : REPEAT '{' Instructions '}' UNTIL '(' Cond ')'"
    ciclo = "ciclo" + str(p.parser.nr)
    p.parser.nr += 1
    p[0] = ciclo + ":\n" + p[3] + p[7] + "jz " + ciclo + "\n"

def p_PrintIdNum(p):
    "Print : PRINT '(' IdNum ')' "
    p[0] = p[3] + "writei\n"

def p_PrintArray(p):
    "Print : PRINT '(' Array ')' "
    p[0] = p[3] + "loadn\nwritei\n"

def p_ReadID(p):
    "Read : READ '(' ID ')' "
    (_,offset,_,_) = p.parser.table[p[3]]
    p[0] = "read\natoi\nstoreg " +  str(offset) + "\n"


def p_ReadArray(p):
    "Read : READ '(' Array ')' "
    p[0] = p[3] + "read\natoi\nstoren\n"


def p_Cond_Cond(p):
   "Cond : Cond OR Cond2"
   p[0] = p[1] + p[2] + p[3]

def p_Cond_Cond2(p):
    "Cond : Cond2"
    p[0] = p[1]

def p_Cond2(p):
    "Cond2 : Cond2 AND Cond3"
    p[0] = p[1] + p[3] + "mul\n"
```

```
def p_Cond2_Cond3(p):
    "Cond2 : Cond3"
    p[0] = p[1]

def p_Cond3_Not(p):
    "Cond3 : NOT Cond"
    p[0] = p[2] + "not\n"

def p_Cond3_ExpR(p):
    "Cond3 : ExpRelacional"
    p[0] = p[1]

def p_Cond3(p):
    "Cond3 : '(' Cond ')'"
    p[0] = p[1]

def p_ExpRelacional_Bigger(p):
    "ExpRelacional : Exp '>' Exp"
    p[0] = p[1] + p[3] + "sup\n"


def p_ExpRelacional_Lower(p):
    "ExpRelacional : Exp '<' Exp"
    p[0] = p[1] + p[3] + "inf\n"


def p_ExpRelacional_BiggerEqual(p):
    "ExpRelacional : Exp '>' '=' Exp"
    p[0] = p[1] + p[4] + "supeq\n"


def p_ExpRelacional_LowerEqual(p):
    "ExpRelacional : Exp '<' '=' Exp"
    p[0] = p[1] + p[4] + "infeq\n"


def p_ExpRelacional_Diff(p):
    "ExpRelacional : Exp '!' '=' Exp"
    p[0] = p[1] + p[4] + "equal\nnot\n"


def p_ExpRelacional_Equal(p):
    "ExpRelacional : Exp '=' '=' Exp"
    p[0] = p[1] + p[4] + "equal\n"


def p_ExpRelacional(p):
    "ExpRelacional : Exp"
    p[0] = p[1]

def p_ExpPlus(p):
    "Exp : Exp '+' Termo"
    p[0] = p[1] + p[3] + "add\n"
```

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

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

def p_TermoMul(p):
    "Termo : Termo '*' Fator"
    p[0] = p[1] + p[3] + "mul\n"

def p_TermoMod(p):
    "Termo : Termo '%' Fator"
    p[0] = p[1] + p[3] + "mod\n"

def p_TermoFator(p):
    "Termo : Fator"
    p[0] = p[1]

def p_FatorExp(p):
    "Fator : '(' Exp ')' "
    p[0] = p[2]

def p_FatorIdNum(p):
    "Fator : IdNum "
    p[0] = p[1]

def p_FatorArray(p):
    "Fator : Array "
    p[0] = p[1]

def p_IdNum(p):
    "IdNum : Num"
    p[0] = p[1]

def p_IdNumId(p):
    "IdNum : Id"
    p[0] = p[1]

def p_Num(p):
    "Num : NUM "
    p[0] = "pushi " + p[1] + "\n"

def p_Id(p):
    "Id : ID "
    (_,offset,_,_) = p.parser.table[p[1]]
    p[0] = "pushg " + str(offset) + "\n"

def p_ArrayN(p):
    "Array : ArrayNum"
    p[0] = p[1]
```

```
def p_ArrayI(p):
    "Array : ArrayId"
    p[0] = p[1]

def p_ArrayNum(p):
    "ArrayNum : ID '[' NUM ']'"
    (_,offset,_,_) = p.parser.table[p[1]]
    p[0] = "pushgp\npushi " + str(offset) +"\npadd\npushi " + p[3] + "\n"

def p_ArrayID(p):
    "ArrayId : ID '[' ID ']'"
    (_,of1,_,_) = p.parser.table[p[1]]
    (_,of2,_,_) = p.parser.table[p[3]]
    p[0] = "pushgp\npushi " + str(of1) +"\npadd\npushg " + str(of2) + "\n"


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



#Build the parser
parser = yacc.yacc()
parser.table = dict()
parser.offset = 0
parser.nr = 1

#Read input and parse it by line
import sys
linhas=""
for linha in sys.stdin:
    linhas += linha
parser.success = True
result = parser.parse(linhas)

if parser.success:
    print(result)
    #print("Frase Válida reconhecida.")
else:
    print("Frase inválida. Corrija e tente de novo...")
```

Resultados na Máquina Virtual

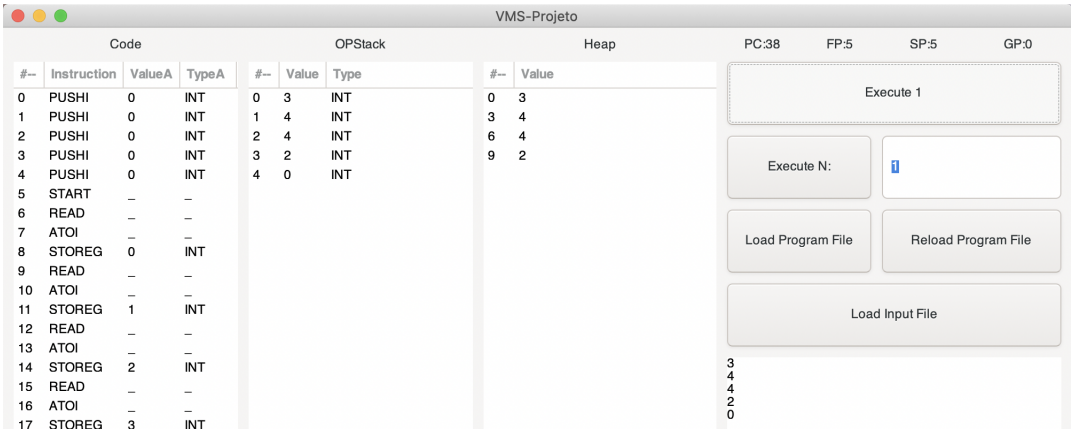

Figura A.1: Resultados obtidos sobre **Leitura de 4 números e possível quadrado**

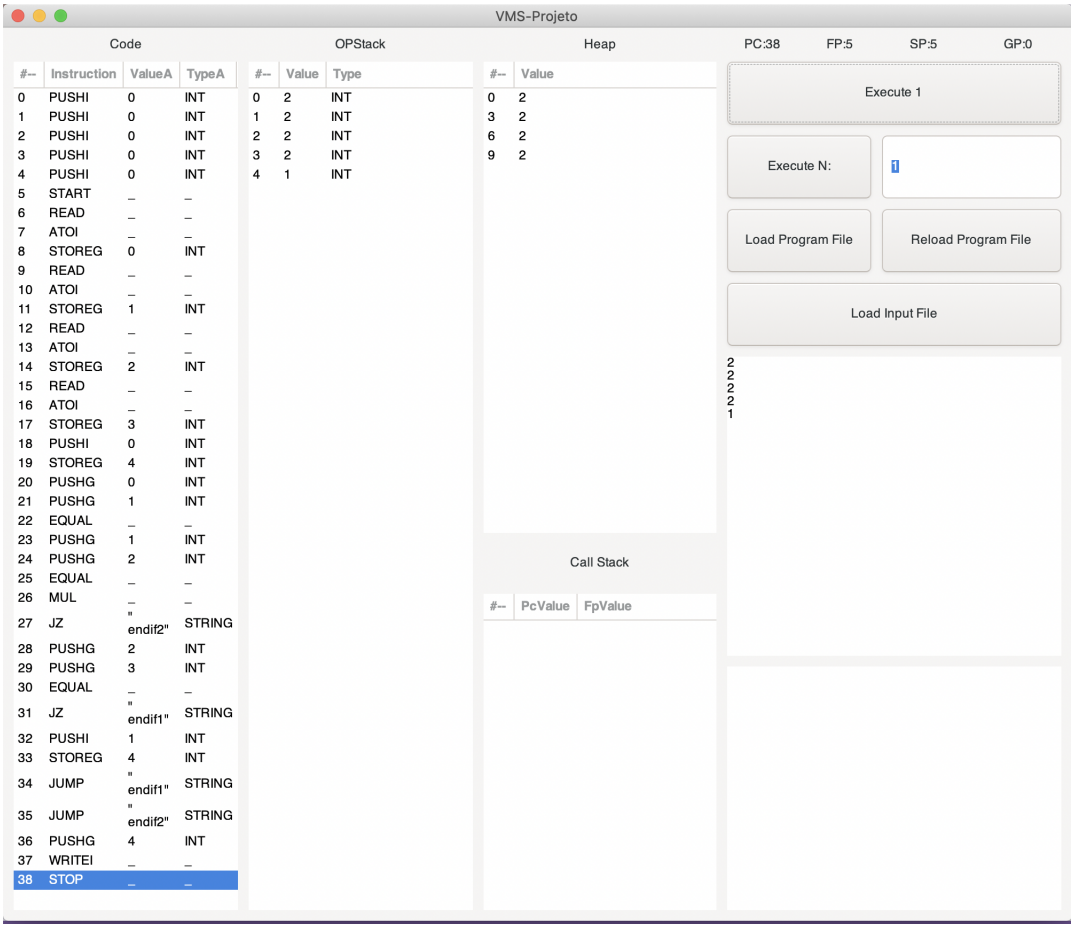

Figura A.2: Resultados obtidos sobre **Leitura de 4 números e possível quadrado**

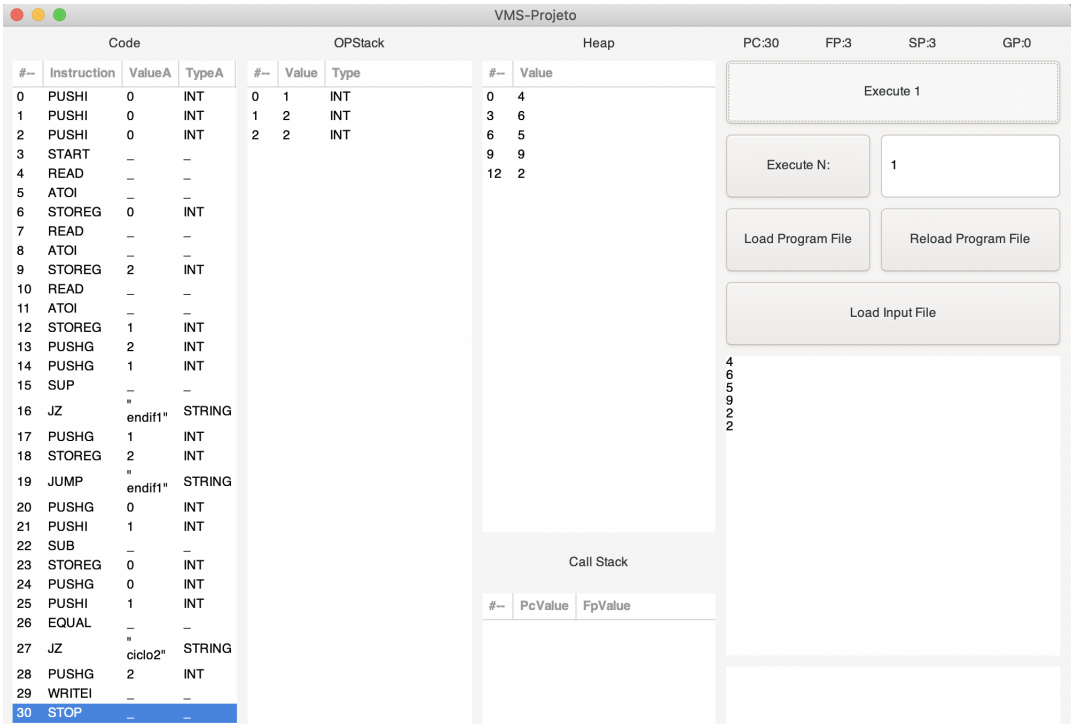

Figura A.3: Resultados obtidos de **Ler um inteiro N, depois ler N números e escrever o menor deles**

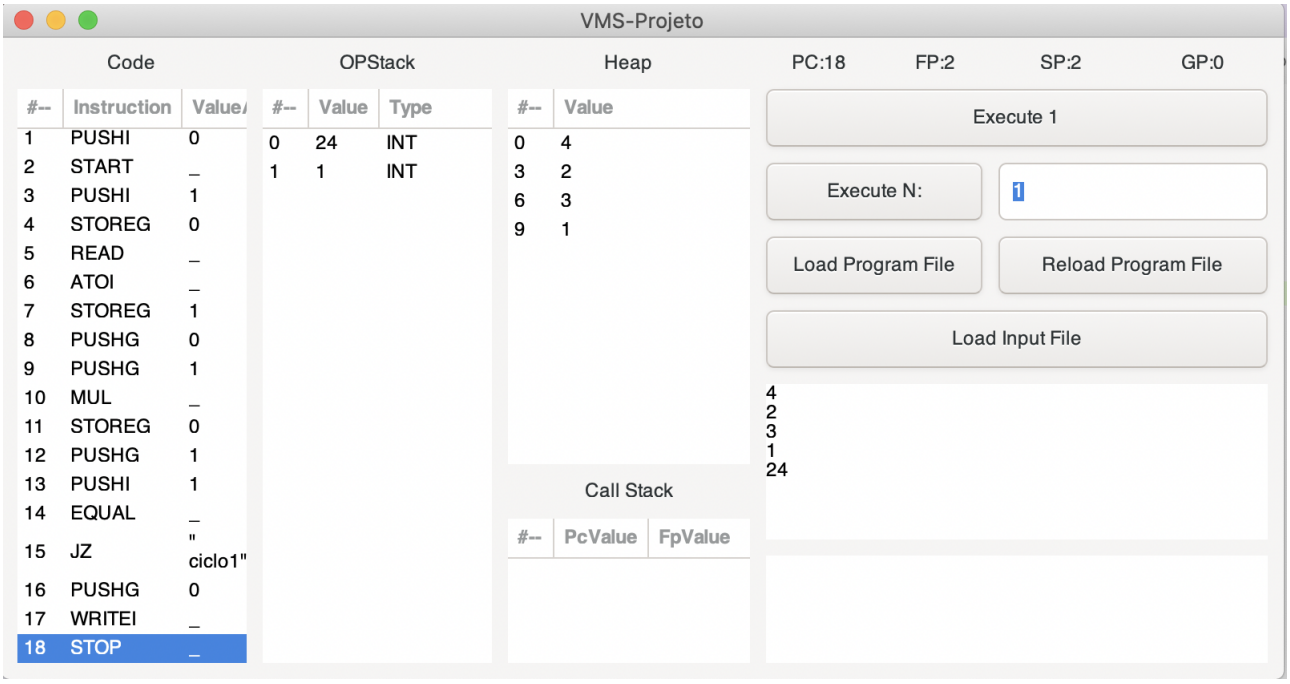

Figura A.4: Resultados obtidos de **Ler N (constante do programa) números e calcular e imprimir o seu produtório.**

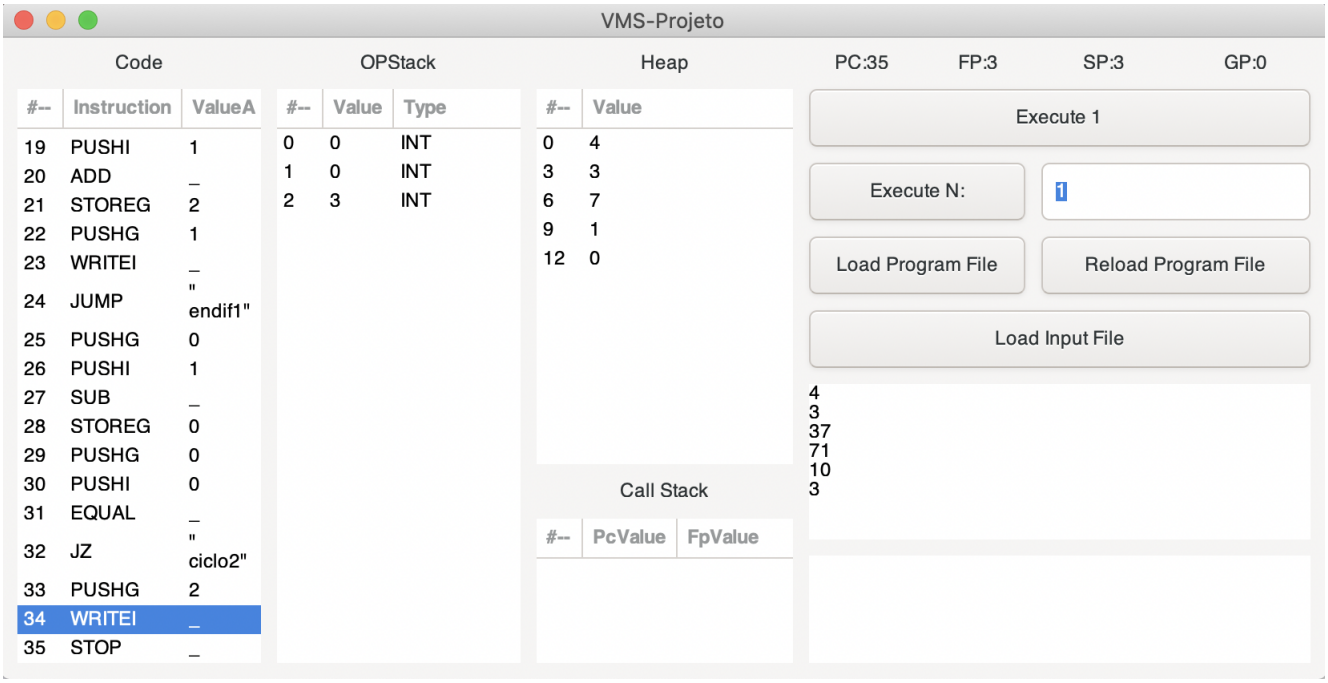

Figura A.5: Resultados obtidos de **Contar e imprimir os números impares de uma sequência de números naturais.**

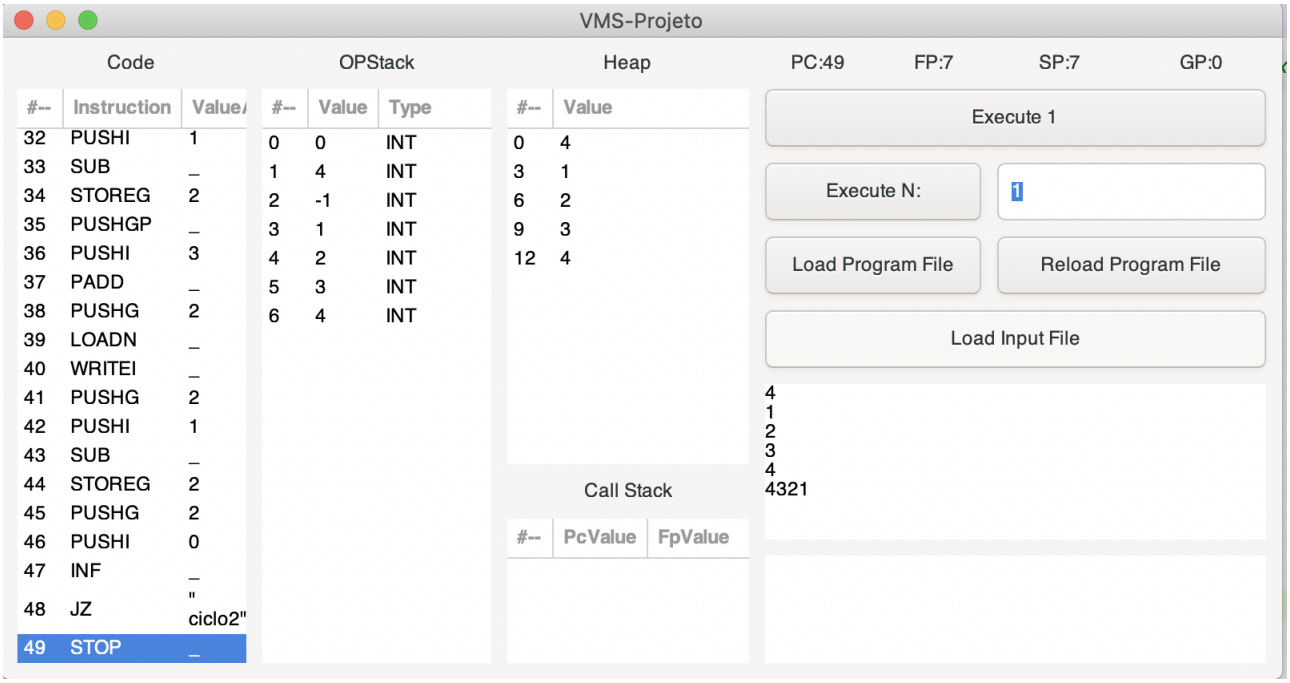

Figura A.6: Resultados obtidos de **Ler e armazenar N números num array; imprimir os valores por ordem inversa.**