

Processamento de Linguagens e Compiladores
(3º ano de LCC)

**Desenvolvimento de Linguagem de
Programação**

TP3

Grupo 14

Artur Queiroz
A77136

Rafael Fernandes
A78242

Rafaela Pinho
A77293

15 de Janeiro de 2018

Resumo

Neste relatório apresentamos a linguagem que criamos, o Galo, e o compilador que gera o código para a Máquina Virtual VM.

Conteúdo

1	Introdução	2
2	Galo e Compilador	3
2.1	Descrição informal do problema	3
2.2	Especificação dos requisitos	3
2.3	Expressões regulares	4
2.4	A nossa linguagem	5
2.4.1	Galo	5
3	Codificação e Testes	8
3.1	Problemas de implementação, Decisões e Alternativas	8
3.1.1	Problemas de implementação	8
3.1.2	Decisões	8
3.2	Testes realizados e Resultados	9
3.2.1	Exemplos escritos na nossa linguagem	9
3.2.2	Resultados	12
4	Conclusão	17
A	Imagens	18
B	Código do Programa	22

Capítulo 1

Introdução

Neste relatório apresentamos o último trabalho da unidade curricular "Processamento de Linguagens e Compiladores". Este consiste em desenvolver um processador de linguagens usando o método da tradução dirigida pela sintaxe. Também devemos desenvolver um compilador que gera o código para uma máquina de stack virtual. Utilizaremos a ferramenta Yacc para gerar compiladores baseados em gramáticas tradutoras.

Como todos os outros trabalhos, este também tem como objetivo aumentar a experiência no uso do ambiente Linux, da linguagem C e ferramentas de apoio à programação.

Este relatório está dividido em 4 partes. No primeiro capítulo encontramos a parte introdutório a este trabalho, onde explicamos no que consiste. No capítulo 2 (dois) apresentamos os requisitos e a linguagem que criamos. No terceiro mostramos as decisões e alguns teste efectuados. No último temos a conclusão e os objetivos do trabalho futuro.

Capítulo 2

Galo e Compilador

2.1 Descrição informal do problema

Neste trabalho foi pedido para criarmos uma linguagem de programação imperativa e desenvolver um compilador para a linguagem criada.

Na linguagem as declarações de variáveis devem ser colocadas no início do programa, não pode haver re-declarações e não se pode usar variáveis sem estar declaradas primeiro. Caso não seja atribuído um valor à variável depois da declaração, esta ficará com o valor zero se for um inteiro, se for um float ficará 0.0 e se for string ficará "".

O compilador deve gerar o código assembly para a Máquina Virtual VM.

2.2 Especificação dos requisitos

Para este trabalho a linguagem que criamos tem de conter os seguintes requisitos:

1. Declarar e manusear variáveis atômicas do tipo inteiro e estruturas do tipo array de inteiros.
2. Ler do standard input e escrever no standard output.
3. Fazer instruções básicas como a atribuição de expressões a variáveis.
4. Definir e invocar subprogramas sem parâmetros mas que possam retornar um resultado atômico.
5. Efetuar instruções para controlo do fluxo de execução — condicional e cíclica — que possam ser aninhadas.

Também deverá conter um conjunto de testes (escritos na nossa linguagem) que tem de ter no mínimo os 6 exemplos seguintes:

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

- ii) Ler um inteiro N, depois ler N números e escrever o menor deles.
- iii) Ler N (constante do programa) números e calcular e imprimir o seu produto.
- iv) Contar e imprimir os números ímpares de uma sequência de números naturais.
- v) Ler e armazenar os elementos de um vetor de comprimento N; imprimir os valores por ordem decrescente após fazer a ordenação do array por trocas diretas.
- vi) Ler e armazenar N números num array; imprimir os valores por ordem inversa

2.3 Expressões regulares

As expressões regulares usadas foram:

1. #.*\$
2. \"[^^]*\"
3. []+e[]+
4. []+ou[]+
5. []+sin[]+
6. []+cos[]+
7. ==
8. \<=
9. \>=
10. !=
11. [=;{}(),<>!\\+\\-*\\/\\%\\[\\]]
12. se|SE
13. senao|SENAO
14. enq|ENQ
15. return
16. [0-9]+\\. [0-9]+
17. int|string|float

18. `-?[0-9]+`
19. `[a-zA-Z][a-zA-Z0-9]*`
20. `[\t\n]`

2.4 A nossa linguagem

2.4.1 Galo

Como já referido em cima, foi-nos pedidos para criar uma linguagem de programação. Decidimos chamar de Galo por ser um símbolo típico de Portugal, e atribuímos *.gal para a extensão. Para a construção da nossa linguagem fomos inspirados pela pseudo linguagem usada nas aulas pelos professores. Para a definir utilizamos uma gramática independente do contexto, em que tomamos certas decisões que serão especificadas mais à frente.

O Galo reconhece os seguintes tipos: números inteiros (int), números decimais (float) e sequência de caracteres (string). A linguagem usa os habituais símbolos de comparação, como `<=`, `>=`, `<`, `>`, `==` e `!=`. Utiliza o "e" e o "ou" como símbolos de operadores lógicos.

Como a nossa linguagem é muito parecida ao C, para fazer o "ite" utilizamos o "se" e o "senao" (tanto minúsculo como maiúsculo) para o "while" usamos o "enq" ou "ENQ".

Existe as funções "ler?()" e "escrever?()" que são, respetivamente, a função de leitura no teclado e de escrita no ecrã. (? = i ou s ou f, i-inteiro, s-string, f- float)

A nossa linguagem está definida pela seguinte GIC:

```

1 ProgG: ProgG Se
2   | ProgG Enq
3   | ProgG Atrib ';'
4   | ProgG VAR '=' Expr ';'
5   | ProgG VAR '[' Expr ']' '=' Expr ';'
6   | ProgG CriaFun
7   | ProgG Funcao ';'
8   | ProgG ';'
9   | ProgG COM
10  | %empty

11 ProgF: ProgF Se
12   | ProgF Enq
13   | ProgF Atrib ';'
14   | ProgF VAR '=' Expr ';'
15   | ProgF VAR '[' Expr ']' '=' Expr ';'
16   | ProgF Funcao ';'
17   | ProgF ';'
18   | ProgF COM
19   | ProgF RETURN Expr ';'
20   | %empty

```

```

21 Prog: Prog Se
22     | Prog Enq
23     | Prog Atrib ';'
24     | Prog VAR '=' Expr ';'
25     | Prog VAR '[' Expr ']' '=' Expr ';'
26     | Prog Funcao ';'
27     | Prog ';'
28     | Prog COM
29     | %empty

30 Funcao: VAR Lexpr

31 CriaFun: TIPO VAR '('
32         | Ltipo '{' ProgF '}'

33 Atrib: TIPO VAR
34         | TIPO VAR '[' Expr ']'
35         | Igual

36 Igual: TIPO VAR '='
37         | TIPO VAR '[' Expr ']' '='
38         | Igual Expr

39 Lexpr: '(' ')'
40         | '(' Eexpr ')'

41 Eexpr: Expr
42         | Eexpr ',' Expr

43 Ltipo: ')'
44         | Etipo ')'

45 Etipo: TIPO VAR
46         | Etipo ',' TIPO VAR

47 Se: SE Cond
48     | Se '{' Prog '}' SENAO
49     | Se '{' Prog '}'

50 Enq: ENQ
51     | Enq Cond
52     | Enq '{' Prog '}'

53 Cond: NUM
54     | '(' Expr EQ Expr ')'
55     | '(' Expr NEQ Expr ')'
56     | '(' Expr '<' Expr ')'
57     | '(' Expr '>' Expr ')'
58     | '(' Expr LEQ Expr ')'
59     | '(' Expr GEQ Expr ')'
60     | '(' Cond E Cond ')'
61     | '(' Cond OU Cond ')'
62     | '!' Cond

63 Sexpr: VAR
64         | NUM
65         | FLOAT

```



```

66      | VAR '[' Expr ']'
67      | Funcao
68      | STR

69 Expr: '(' Expr '+' Expr ')'
70      | '(' Expr '-' Expr ')'
71      | '(' Expr '*' Expr ')'
72      | '(' Expr '/' Expr ')'
73      | '(' Expr '%' Expr ')'
74      | COS Expr
75      | SIN Expr
76      | '(' Expr ')'
77      | Sexpr

```

Capítulo 3

Codificação e Testes

3.1 Problemas de implementação, Decisões e Alternativas

3.1.1 Problemas de implementação

Como em todos os trabalhos tivemos alguns contratempos, dos quais muitos foram superados.

Quando estavamos a passar a nossa linguagem para uma gramática tradutora tivemos alguns problemas com o código assembly da máquina virtual, mas nada que não se resolvesse com um bocadinho de paciência e trabalho de grupo. Tivemos problemas com a implementação de vetores, pois não conseguimos compreender a forma de utilização correcta do comando load.

3.1.2 Decisões

- 1) O "e"(&&) está definida pela multiplicação e o "ou"(||) pela adição.

Tabela 3.1: Tabela do E

* (E)	0	1
0	0	0
1	0	1

Tabela 3.2: Tabela do OU

+(OU)	0	1
0	0	1
1	1	2

- 2) Não se pode declarar mais do que uma variável numa linha, ou seja todas as declarações são individuais.

Exemplo:
int a = 2 , c = 0;
terá de ser:
int a = 2;
int c = 0;

- 3) Não se pode fazer "return" dentro dos Se's e dos Enq's.
- 4) Nas expressões numéricas, as operações binárias têm de estar sempre dentro de parênteses.
Exemplo:
int a = (1+(2*3));
- 5) Todo o código dentro de se's, enq's e funções começa em '{' e acaba em '}'.

3.2 Testes realizados e Resultados

3.2.1 Exemplos escritos na nossa linguagem

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

```
int a = leri();
int b = leri();
int c = leri();
int d = leri();

se ((a==b) e ((b==c) e (c==d))) {
    escrevers("É um quadrado\n");
}
senao {
    escrevers("Não é um quadrado\n");
}
```

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

```
escrevers("Escreva o número de elementos do array:\n");
int N = leri();

int i = 0;
int a = 0;
int res = 0;

se (N>0){
    res = leri();
    i = 1;
    enq (i < N){
        a = leri();
```

```

        se (res > a){
            res = a;
        }

        i = (i+1);
    }

    escrevers("0 menor número foi o ");
    escreveri(res);
    escrevers("\n");
}
senao{
    escrevers("Não leu nenhum número\n");
}

```

3. Ler N (constante do programa) números e calcular e imprimir o seu produto.

```

    escrevers("Vão se ler 5 números\n");

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

    enq(i < N){

        r = (r * leri());

        i = (i + 1);
    }

    escrevers("0 produto desta sequencia de 5 números é ");
    escreveri(r);
    escrevers("\n");

```

4. Contar e imprimir os números ímpares de uma sequência de números naturais.

```

    escrevers("Digitar uma sequência de números, termina quando for zero\n");

    int cont = 0;
    int i = leri();

    enq(i != 0){
        se ((i % 2) == 1){
            cont = (cont + 1);
            escreveri(i);
            escrevers(" \n");
        }
        i = leri();
    }

```

```

escrevers("Foram lidos ");
escreveri(cont);
escrevers(" ímpares \n");

```

5. Ler e armazenar os elementos de um vetor de comprimento N; imprimir os valores por ordem decrescente após fazer a ordenação do array por trocas diretas.

```

int troca(int v, int i, int j){
    int k = v[i];
    v[i] = v[j];
    v[j] = k;
    return 0;
}

int ordena(int[] v, int N){
    int i = 0; #inicio
    int j = 0; #procura
    int m ;    #pos do menor

    enq (i<(N-1)){
        j = (i + 1);
        m = i;
        enq (j<N){
            se(v[j]>v[m]){
                m = j;
            }
            j = (j + 1);
        }
        troca(v,i,m);
        i = (i + 1);
    }
    return 0;
}

int N = leri();
int i = 0;
int v[N];

enq(i<N){
    v[i] = leri();
    i = (i+1);
}

ordena(v, N);

i = 0;

enq(i<N){
    escreveri(v[i]);
    escrevers("\n");
}

```

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

```

int N = leri();
int i = 0;
int a = 0;
int v[N];

enq (i < N){
    v[i] = leri();
    i = (i + 1);
}

enq(i > 0){
    a = v[(i-1)];
    escreveri(a);
    escrevers("\n");
    i = (i-1);
}

```

3.2.2 Resultados

Depois de executarmos os comandos seguintes:

```
$ flex -o galo.c galo.l
```

```
$ yacc -d -v galo.y
```

```
$ gcc -o galo y.tab.c -lm
```

compilamos todos os nossos exemplos.

Obtivemos o seguinte resultado:

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

```

start
pushi 0
read
atoi
storeg 0
pushi 0
read
atoi
storeg 1
pushi 0
read
atoi
storeg 2
pushi 0
read
atoi
storeg 3
pushg 0
pushg 1
equal
pushg 1
pushg 2
equal
pushg 2

```

```

pushg 3
equal
mul
mul
jz fimse0
pushs "É um quadrado\n"
writes
jump fimse1
fimse0:
pushs "Não é um quadrado\n"
writes
fimse1:
stop

```

Após abrir o ficheiro na máquina virtual, se atribuirmos, por exemplo, o número 4 ao a, b, c, d o programa retorna **"É um quadrado"** (Figura A.1). Se atribuirmos valores diferentes ao a, b, c, d o programa diz **"Não é um quadrado"** (Figura A.2).

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

```

start
pushs "Escreva o número de elementos do array:\n"
writes
pushi 0
read
atoi
storeg 0
pushi 0
pushi 0
storeg 1
pushi 0
pushi 0
storeg 2
pushi 0
pushi 0
storeg 3
pushg 0
pushi 0
equal
pushi 0
equal
jz fimse0
read
atoi
storeg 3
pushi 1
storeg 1
enq0:
pushg 1
pushg 0
inf
jz fimenq0
read
atoi
storeg 2

```

```

pushg 3
pushg 2
sup
jz fimse1
pushg 2
storeg 3
fimse1:
pushg 1
pushi 1
add
storeg 1
jump enq0
fimenq0:
pushs "0 menor número foi o "
writes
pushg 3
writei
pushs "\n"
writes
jump fimse2
fimse0:
pushs "Não leu nenhum número\n"
writes
fimse2:
stop

```

Definimos que o array tem 3 elementos e depois digitamos valores, por exemplo o 7, 5 e o 8. O resultado deste programa é o menor desses elementos, que é o 5 (figura A.3). Se o array tivesse 0 elementos o programa diz que "Não leu nenhum número" (figura A.4).

3. Ler N (constante do programa) números e calcular e imprimir o seu produto.

```

start
pushs "Vão se ler 5 números\n"
writes
pushi 0
pushi 5
storeg 0
pushi 0
pushi 0
storeg 1
pushi 0
pushi 1
storeg 2
enq0:
pushg 1
pushg 0
inf
jz fimenq0
pushg 2
read
atoi
mul

```



```

storeg 2
pushg 1
pushi 1
add
storeg 1
jump enq0
fimenq0:
pushs "O produtório desta sequencia de 5 números é "
writes
pushg 2
writei
pushs "\n"
writes
stop

```

Atribuímos 5 como o número de elementos do array. Depois digitamos a sequência 3,4,5,6,8 de e o resultado do programa de 2880 que é o produtório da sequência (figura A.5).

4. Contar e imprimir os números ímpares de uma sequência de números naturais.

```

start
pushs "Digitar uma sequência de números, termina quando for zero\n"
writes
pushi 0
pushi 0
storeg 0
pushi 0
read
atoi
storeg 1
enq0:
pushg 1
pushi 0
equal
pushi 0
equal
jz fimenq0
pushg 1
pushi 2
mod
pushi 1
equal
jz fimse0
pushg 0
pushi 1
add
storeg 0
pushg 1
writei
pushs " \n"
writes
fimse0:
read

```

```
atoi
storeg 1
jump enq0
fimenq0:
pushs "Foram lidos "
writes
pushg 0
writei
pushs " impares \n"
writes
stop
```

Começa por aparecer a mensagem para digitarmos uma sequência de números. Digitando os valores 3, 1, 2, 6, 5 e 0. À medida que vamos inserindo os números vão aparecendo os que são ímpares. Depois quando digitamos o zero aparece a quantidade de números ímpares que foram lidos. (figura A.6).

Capítulo 4

Conclusão

Este trabalho abrangeu maior parte da matéria lecionada ao longo do semestre. Através do nosso conhecimento em GIC's e sobre o gerador Yacc e Flex criamos um compilador para converter a nossa linguagem em pseudo-código para a Máquina Virtual VM.

Não conseguimos acabar todas as tarefas propostas, o exemplo 5 e 6 não estão a funcionar devido aos vetores. Como referido no capítulo 3 tivemos dificuldades a implementar essa estrutura.

Como trabalho futuro gostaríamos de implementar os vetores e adicionar mais elementos à nossa linguagem, como a parte gráfica.

Apêndice A

Imagens

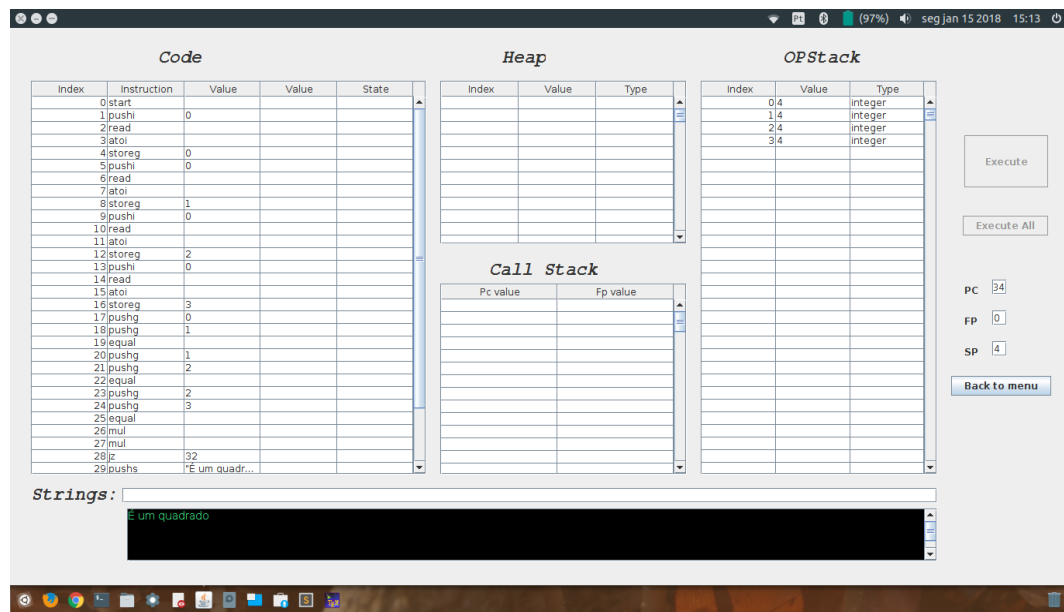


Figura A.1: Verifica se é um quadrado

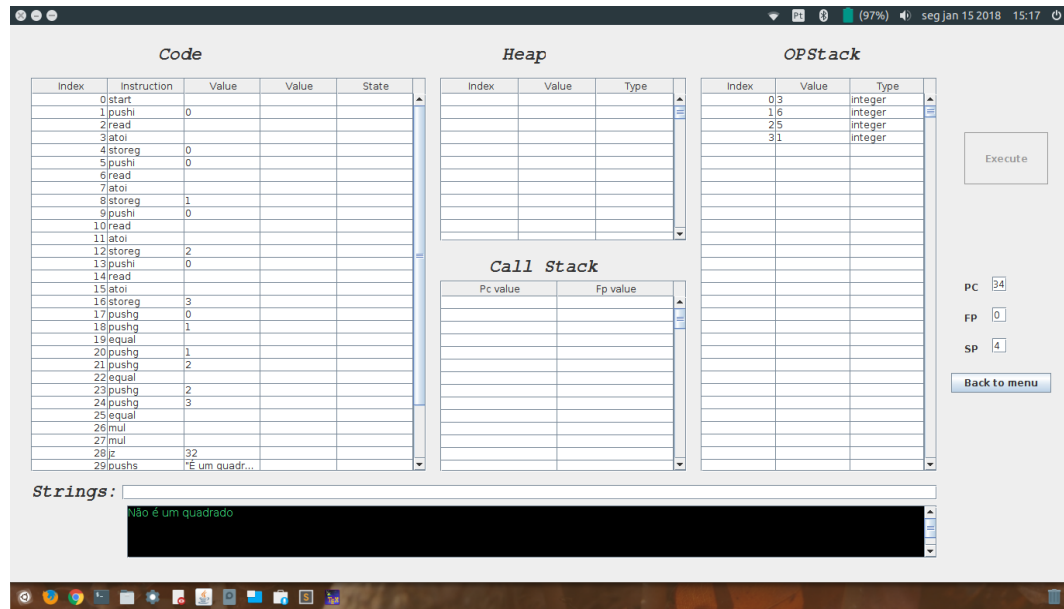


Figura A.2: Verifica que não é um quadrado

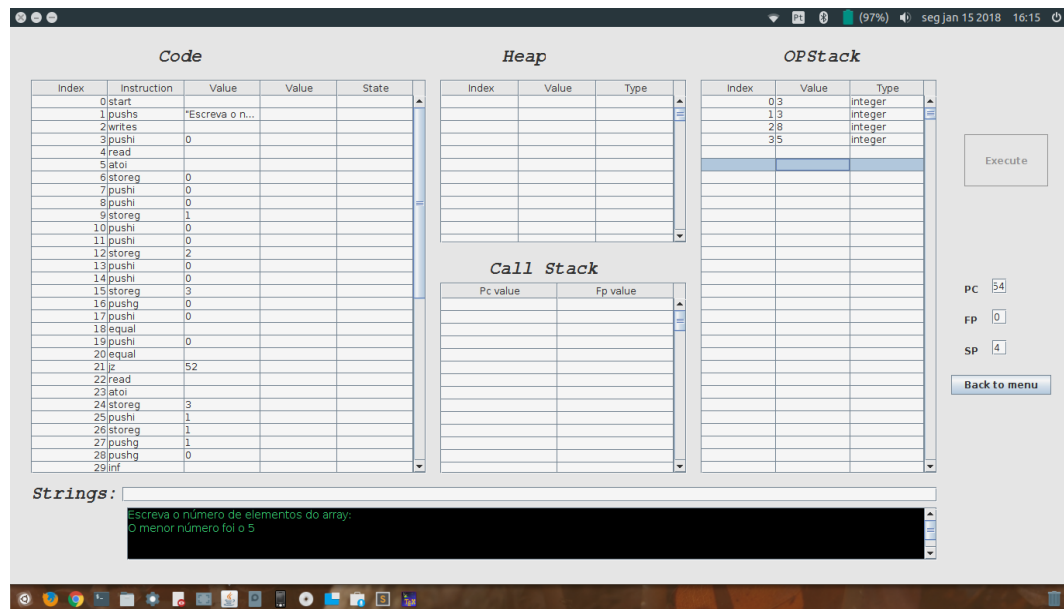
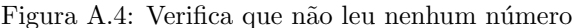


Figura A.3: Diz qual é o menor número



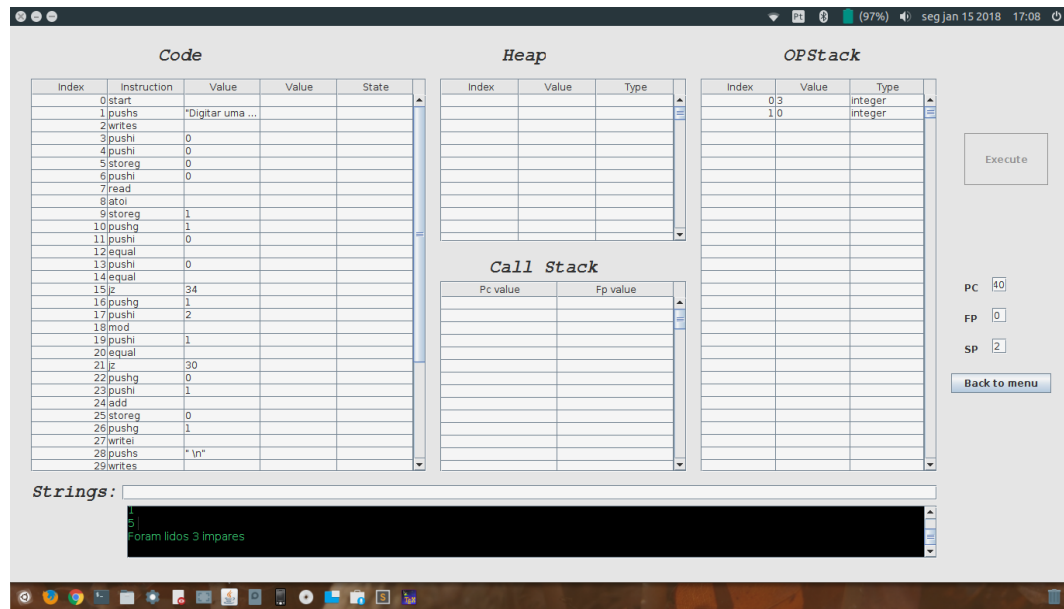


Figura A.6: Mostra quantos impares foram lidos

Apêndice B

Código do Programa

Só o ficheiro galo.l

```
%{
#include "y.tab.h"
%}

%%
#.*$                                { return(COM); }
\[^[^"]*\\"                        { yylval.s = strdup(yytext); return(STR); }
[ ]+e[ ]+                          { return(E); }
[ ]+ou[ ]+                         { return(OU); }
[ ]+sin[ ]+                        { return(SIN); }
[ ]+cos[ ]+                        { return(COS); }
==                                 { return(EQ); }
\<=                                { return(LEQ); }
\>=                                { return(GEQ); }
!=                                 { return(NEQ); }
[=;{}()<>!\+\\-\\*\\/%%\\[\\]]   { return(yytext[0]); }
se|SE                              { return(SE); }
senao|SENAO                       { return(SENAO); }
enq|ENQ                           { return(ENQ); }
return                            { return(RETURN); }
[0-9]+\.[0-9]+                    { yylval.f = atof(yytext); return(FLOAT); }
int|string|float                  { yylval.s = strdup(removeEspacos(yytext)); return(TIPO); }
-?[0-9]+                          { yylval.i = atoi(yytext); return(NUM); }
[a-zA-Z][a-zA-Z0-9]*             { yylval.s = strdup(yytext); return(VAR); }
[ \t\n]                           { ; }
.                                  { printf("Caracter invalido %c\n", yytext[0]); }

%%

int yywrap(){
    return(1);
}
```