

Compilador para VM usando Yacc e Flex
Processamento de Linguagens (3º ano de Curso)
Trabalho Prático 2
Relatório de Desenvolvimento

José Silva
(A74601)

Pedro Cunha
(A73958)

Gonçalo Moreira
(A73591)

15 de Junho de 2017

Resumo

Documentação do terceiro trabalho prático da unidade curricular de "Processamento de Linguagens", o principal foco incide sobre a criação de um compilador baseado numa gramática tradutora. Inicialmente é idealizada uma linguagem de programação simples, capaz de dar resposta a uma série de requisitos base. Demonstrando e documentando a solução proposta pelo grupo de trabalho para o problema em concreto, termina-se o relatório com uma análise argumentativa sobre a eficiência dessa mesma solução.

Conteúdo

1	Introdução	2
2	Análise e Especificação	3
2.1	Descrição informal do problema	3
2.2	Especificação do Requisitos	3
2.2.1	Dados	3
3	Concepção/desenho da Resolução	5
3.1	Estruturas de Dados	6
3.2	Algoritmos	7
4	Codificação e Testes	9
4.1	Alternativas, Decisões e Problemas de Implementação	9
4.2	Testes realizados e Resultados	9
5	Conclusão	20
A	Código do Programa	21

Capítulo 1

Introdução

O terceiro trabalho prático da unidade curricular de "Processamento de Linguagens" tem como principal objetivo o desenvolvimento de um processador de uma linguagem segundo um método de tradução dirigida pela sintaxe e suportado numa gramática. Para além disso, pretende-se desenvolver um compilador através da geração de código para uma máquina de stack virtual.

Inicialmente é idealizada uma linguagem de programação simples, capaz de dar resposta a uma série de requisitos base. Posteriormente, é desenvolvido o compilador capaz de gerar pseudocódigo Assembly na máquina virtual fornecida, com base na GIC desenvolvida e recorrendo às ferramentas Flex e Yacc.

Estrutura do Relatório

No capítulo 1 faz-se uma pequena introdução ao problema e às ferramentas utilizadas para a resolução deste. Para além disso, é descrita de uma forma breve a estrutura do relatório.

No capítulo 2 faz-se uma análise breve mas mais detalhada do problema em questão.

No capítulo 3 é descrito de uma forma sumariada o procedimento utilizado para solucionar as várias questões propostas pelos enunciados.

No capítulo 4 são apresentados alguns testes e respetivos resultados para comprovar o respectivo funcionamento das soluções apresentadas.

Finalmente, no capítulo 5 termina-se o relatório com uma síntese do que foi dito, as conclusões e o trabalho futuro.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

O problema proposto pelo enunciado do trabalho prático tem como base uma linguagem de programação, idealizada pelo grupo de trabalho, que deverá permitir declarar e manusear variáveis, atômicas e estruturadas, realizar operações e instruções (algorítmicas e de controlo de fluxo de execução) básicas, bem como, ler do standard input e output.

Para além disso, são consideradas alguns aspectos comuns deste tipo de linguagens de programação. Após a realização das análises léxicas e sintáticas, acompanhadas pelas respectivas ações a tomar, pretende-se gerar um ficheiro com instruções Assembly corretamente ordenadas e capazes de produzirem o resultado final pretendido.

2.2 Especificação do Requisitos

2.2.1 Dados

Antes da criação da GIC foi especificado, pelo grupo de trabalho, alguns requisitos base que o formato da linguagem imperativa deveria de ter. Assim, foram tomadas as seguintes decisões:

1. As variáveis devem ser declaradas no início do programa. A primeira declaração deve ser sempre antecedida por "VAR:". Qualquer declaração deve ser precedida por ";". É possível declarar variáveis atômicas mas igualmente variáveis estruturadas. No que diz respeito às últimas, as variáveis estruturadas de dimensão um, devem ser declaradas com o seu tamanho entre parênteses e as de dimensão dois com o número de linhas entre brackets e o número de colunas entre parênteses.
2. Finalizadas todas as declarações de variáveis, o início das várias instruções que compõe o programa deve ser especificado com "START:". Após esta declaração, podem ser especificadas vários tipos de instruções.
3. No que diz respeito às instruções que permitem atribuições a determinadas variáveis, estas devem ter o seguinte formato: variável -i" expressão.
4. Relativamente à interação com o standard input e output, deve ser feita utilizando o formato: Pread(x) e Pprint(x), respectivamente, em que x pode ser uma variável atômica ou um array de uma ou duas dimensões.
5. Retratando o controlo de fluxo de execução, as instruções condicionais podem ser especificadas através de "if" e "else", sendo o seu significado semelhante ao utilizado em praticamente todas as linguagens de programação

imperativas atuais. Após estas palavras chave, deve ser incluído uma bracket de abertura e no final das instruções uma bracket de fecho. Finalmente, para especificar ciclos de execução, podem ser utilizada a palavra "while", onde o processo é semelhante ao descrito anteriormente.

Capítulo 3

Concepção/desenho da Resolução

No decorrer da definição da linguagem imperativa, a qual apelidamos de Plang, foi construída a seguinte GIC:

gramatica.txt

```
1 G = <T,N,S,P>
2 T = {VAR,':',';','[' ,']','{','}','', START, '-','>','(',' ,')',' V,
3 PREAD, PPRINT, STRING, IF, ELSE, WHILE, '|','&','=','!','<','+',
4 '-','*','/','%','^',' INTGR }
5 N = {Plang,Init,Body,Declare,Variable,Body,Instruction, Instructions,
6 Assignment, Read, Print, Condicional, Opelse, Cyclic, Accumulator, Comparator,
7 Expression, P, Fat, Es}
8 S = {Plang}
9
10 p0: Plang      ->  Init Body
11 p1: Init       ->  VAR ':' Declare
12 p2: Declare    ->  Declare Variable
13 p3: Variable   ->  V ';'
14 p4:           |  V '[' INTGR ']' ';'
15 p5:           |  V '{' INTGR '}' '[' INTGR ']' ';'
16
17 p6: Body       ->  START ':' Instructions ';'
18 p7: Instructions ->  Instructions Instruction
19 p8:           |  Instruction
20
21 p9: Instruction ->  Assignment
22 p10:          |  Read
23 p11:          |  Print
24 p12:          |  Condicional
25 p13:          |  Cyclic
26
27 p14: Assignment ->  V '-' '>' Expression ';'
28 p15:          |  V '[' Expression ']' '-' '>' Expression ';'
29 p16:          |  V '{' Expression '}' '[' Expression ']' '-' '>' Expression ';'
30
31 p17: Read      ->  PREAD '(' V ')' ';'
32 p18:          |  PREAD '(' V { '[' Expression ']' ')' ';'
33 p19:          |  PREAD '(' V '{' { Expression '}' '[' Expression ']' ')' ';'
34
35 p20: Print     ->  PPRINT '(' Expression ')' ';'
36 p21:          |  PPRINT '(' STRING ')' ';'
37
```

```

38 p22: Condicional    -> IF '(' Accumulator ')' '{' Instructions '}' Opelse
39 p23: Opelse        -> ELSE '{' Instructions '}'
40 p24:                |
41
42 p25: Cyclic         -> WHILE '(' Accumulator ')' '{' Instructions '}'
43 p26: Accumulator    -> Comparator '|' '|' Accumulator
44 p27:                | Comparator '&' '&' Accumulator
45 p28:                | Comparator
46
47 p29: Comparator     -> Expression
48 p30:                | Expression '=' '=' Expression
49 p31:                | Expression '!' '=' Expression
50 p32:                | Expression '>' Expression
51 p33:                | Expression '<' Expression
52 p34:                | Expression '>' '=' Expression
53 p35:                | Expression '<' '=' Expression
54
55 p36: Expression     -> Expression '+' P
56 p37:                | Expression '-' P
57 p38:                | P
58
59 p39: P              -> P '*' Fat
60 p40:                | P '/' Fat
61 p41:                | P '%' Fat
62 p42:                | Fat
63
64 p43: Fat            -> Es '^' Fat
65 p44:                | Es
66
67 p45: Es             -> '(' Expression ')'
68 p46:                | INTGR
69 p47:                | '-' INTGR
70 p48:                | V
71 p49:                | '-' V
72 p50:                | V '[' Expression ']'
73 p51:                | '-' V '[' Expression ']'
74 p52:                | V '{' Expression '}' '[' Expression ']'
75 p53:                | '-' V '{' Expression '}' '[' Expression ']'

```

3.1 Estruturas de Dados

De modo a ser possível desenvolver algumas das funcionalidades base descritas acima, foi necessária a definição de estruturas de dados capazes de as suportar. Dado que, como vai ser descrito na secção a seguir, é necessário guardar informação relativamente às variáveis declaradas, o grupo de trabalho utilizou a HashTable disponibilizada pela biblioteca glib e criou a struct definida a seguir:

```

1 typedef struct variable {
2     int stack;
3     int size;
4 } *Var;

```

A definição de uma HashTable que, possui o nome da variável como chave e a struct descrita anteriormente como valor, permite testar se existem ou não re-declarações no código da linguagem e permitiu ainda guardar informação útil

respeitante às variáveis declaradas (Endereço na stack e o tamanho das colunas, para o caso das variáveis estruturadas). Também é declarado um inteiro com o intuito de representar um stack pointer.

Através dos algoritmos que vão ser a seguir descritos é possível perceber a utilidade destas estruturas de dados para a implementação das funcionalidades da linguagem.

Para controlar os ifs, elses e whiles da linguagem foi utilizada uma stack. Desta forma, tratou-se estes casos como LIFO garantindo uma execução correta em ifs, elses e whiles encadeados ou aninhados. A stack tem sempre no topo o último jump utilizado, sabendo assim qual o bloco a definir. Definição da stack:

```
1  #define MAXSTACK 512
2
3  typedef struct stack {
4      int blocks[MAXSTACK];
5      int top;
6  } *Stack;
7
8  int pop(Stack);
9  void push(int, Stack);
```

3.2 Algoritmos

Tendo em conta aquilo que foi descrito nas secções anteriores, podemos descrever de forma resumida como foi resolvido o problema em questão, ou seja, o que deve resultar da tradução da linguagem criada pelo grupo de trabalho para linguagem assembly.

Assim, no que diz respeito à declaração de variáveis de variáveis atómicas, primeiro dá-se uma verificação se já tinham sido declaradas e caso isso não se verifique são empilhadas na stack através de PUSHI X, em que x é o 0 por default. O valor do stack pointer é guardado na estrutura da variável e incrementado logo de seguida. No caso das variáveis estruturais, é usado PUSHN x que resulta em empilhar x vezes o valor 0 na stack. O valor do stack pointer também é guardado, incrementado pelo tamanho, no caso do array, e incrementado pelo produto do número de linhas e colunas, no caso da matriz.

Depois da declaração inicial das variáveis e do corpo do programa, deve ser gerada a instrução de código Assembly "START" que indica o início das instruções. Posteriormente, no que diz respeito à atribuição de valores, no caso de a variável ser atómica, apenas se empilha o valor que se quer atribuir e de seguida faz-se um STOREG x, em que x é o endereço da variável em questão e que se encontra guardado na estrutura de dados descrita na secção anterior.

O processo torna-se mais complexo em relação aos arrays e às matrizes. Em relação aos arrays, o processo inicial passa sempre por empilhar o valor de gp (endereço de base das variáveis globais), o valor correspondente à posição do início do array (em relação ao gp) e, finalmente, o valor correspondente a essa posição somada com o gp (PUSHGP, PUSHI X, PADD). O passo seguinte passa por empilhar o valor desejado a atribuir e a posição desejada no array. Finalmente, através de STOREN, é arquivada na posição (início array + posição desejada) (trazidas para a stack com as instruções anteriores) o valor também trazido para a stack no passo anterior.

Em relação às matrizes, o processo é, em parte, semelhante, diferindo no facto de que para o acesso a $m[i][j]$ é necessário aplicar a seguinte fórmula: $i * (\text{tamanho de cada linha}) + j$, e havendo assim a necessidade de utilizar alguns operadores, como ADD e MUL. Já no que diz respeito à escrita para o standard output, o processo é bastante simples. Para variáveis atómicas apenas é necessário WRITEI. Para strings, primeiro é necessário empilha-las e de seguida um WRITES.

Por outro lado, relativamente ao standard input, deve ser gerado um READ para ler a string do STDIN, ATOI para

a converter essa string num inteiro e STOREG para a armazenar. No caso específico de arrays, o processo passa por a fase inicial da atribuição descrita no parágrafo anterior, seguido das instruções relatadas na linha anterior, com a substituição de STOREG por STOREN.

Para os blocos definidos para ifs, elses e whiles existe uma variável global(inteiro) que garante que não existem blocos com o mesmo nome, porque incrementa sempre que se cria algum e esse número é usado nos nomes dos blocos. No caso particular dos ifs com ou sem else existe uma stack para eles, no início de um if são colocadas as condições, por exemplo EQUAL. De seguida, é efetuado um JZ blocox, e o valor de x é colocado na stack e o pop do mesmo só é efetuado no final do if, que é onde começa o else ou então o resto do código caso não exista else. Assim, quando se faz pop, coloca-se o valor de return do pop no nome do bloco que define o código deste ponto para a frente. No caso de haver um else, existe ainda mais um cuidado a ter. É necessário definir um JUMP blocoy, em que blocoy representa o final do else. Este JUMP é definido antes do bloco do else, estando assim dentro do if e é executado se o if for executado. Este JUMP também utiliza a stack dos ifs, sendo que é feito push depois do JUMP e pop da stack no final do else, começando aí o bloco referenciado pelo JUMP. Assim, o else não é executado quando o if é. Quanto ao caso dos whiles é também utilizada uma stack, que segue a mesma lógica. Depois da condição do while é usada a stack dos ifs com o JZ blocox e push x na stack. No final é feito pop dessa mesma stack utilizando o valor de retorno do pop, que corresponde ao bloco referenciado em JZ. Mas isto não é suficiente para o ciclo. Por isso, existe uma stack para os whiles que define um bloco no início do while mesmo antes de verificar a condição, fazendo push para a stack dos whiles do valor usado nesse bloco. Desta forma, no final do while, pode fazer-se pop dessa mesma stack, retira-se o valor do bloco de início do while e efetua-se JUMP para o mesmo e assim a condição será verificada novamente até falhar.

Capítulo 4

Codificação e Testes

4.1 Alternativas, Decisões e Problemas de Implementação

No que diz respeito à idealização da linguagem imperativa, foi desenvolvida uma linguagem com características específicas e próprias, mas capazes de dar resposta às funcionalidades propostas pelo enunciado do trabalho. Já numa fase posterior à criação da GIC, podemos realçar o facto de termos utilizado ações não apenas no final de uma produção, mas também no meio desta. Apesar do grupo de trabalho nunca ter realizado algo semelhante nas aulas práticas, isto veio permitir uma maior facilidade na resolução do problema.

Na primeira fase do trabalho prático, um dos principais problemas de implementação passou por definir a ordem pelas quais as operações aritméticas deveriam aparecer na gramática formulada. Foi com o auxílio do código desenvolvido nas aulas práticas da unidade curricular, em particular no exercício onde se pretendia desenvolver uma gramática descritiva do modo de funcionamento de uma calculadora, que o grupo de trabalho conseguiu garantir que os operadores com maior prioridade iriam ser executados em primeiro lugar.

Surgiu o típico problema "dangling else", pois inicialmente as opções de if sozinho ou com else causavam conflito na gramática. Existia um conflito reduce/reduce, que foi resolvido acrescentando uma nova produção designada Opelse. Houve também um problema na definição de matrizes e arrays, havia conflito com a forma de representação. Então para representar as linhas da matriz foram usadas chavetas, e para as colunas os parênteses retos como no array.

4.2 Testes realizados e Resultados

Programa que le e armazena N inteiros num array, ordena e imprime de por ordem decrescente

Programas/ordena.plang

```
1  # Programa que le e armazena N inteiros num array, ordena e imprime de por ordem
    decrescente
2  VAR:
3      numeros [64];
4      quantos;
5      aux;
6      i; j;
7  START:
8      Pprint("Quantos inteiros a ordenar?(Max:64)\n");
9      Pread(quantos);
10
```

```

11  i -> 0;
12  while(i < quantos) {
13      Pread(numeros[i]);
14      i -> i + 1;
15  }
16
17  i -> 0;
18  while(i < quantos){
19      j -> i + 1;
20      while(j < quantos){
21          if(numeros[i] > numeros[j]) {
22              aux -> numeros[i];
23              numeros[i] -> numeros[j];
24              numeros[j] -> aux;
25          }
26          j -> j + 1;
27      }
28      i -> i + 1;
29  }
30
31  Pprint("Array em ordem decrescente: \n");
32
33  while(quantos > 0) {
34      quantos -> quantos - 1;
35      Pprint(numeros[quantos]);
36      Pprint(" \n");
37  }
38  ;

```

Assembly gerado:

Assembly/ordena.vm

```

1      pushn 64
2      pushi 0
3      pushi 0
4      pushi 0
5      pushi 0
6  start
7      pushes "Quantos inteiros?(Max:64)\n"
8      writes
9      read
10     atoi
11     storeg 64
12     pushi 0
13     storeg 66
14  bloco1:
15     pushg 66
16     pushg 64
17     inf
18     jz bloco2
19     pushgp
20     pushi 0
21     padd
22     pushg 66
23     read
24     atoi
25     storen
26     pushg 66

```

```

27     pushi 1
28     add
29     storeg 66
30     jump bloco1
31 bloco2:
32     pushi 0
33     storeg 66
34 bloco3:
35     pushg 66
36     pushg 64
37     inf
38     jz bloco4
39     pushg 66
40     pushi 1
41     add
42     storeg 67
43 bloco5:
44     pushg 67
45     pushg 64
46     inf
47     jz bloco6
48     pushgp
49     pushi 0
50     padd
51     pushg 66
52     loadn
53     pushgp
54     pushi 0
55     padd
56     pushg 67
57     loadn
58     sup
59     jz bloco7
60     pushgp
61     pushi 0
62     padd
63     pushg 66
64     loadn
65     storeg 65
66     pushgp
67     pushi 0
68     padd
69     pushg 66
70     pushgp
71     pushi 0
72     padd
73     pushg 67
74     loadn
75     storen
76     pushgp
77     pushi 0
78     padd
79     pushg 67
80     pushg 65
81     storen
82 bloco7:
83     pushg 67
84     pushi 1
85     add

```

```

86     storeg 67
87     jump bloco5
88 bloco6:
89     pushg 66
90     pushi 1
91     add
92     storeg 66
93     jump bloco3
94 bloco4:
95     pushs "Array em ordem decrescente: \n"
96     writes
97 bloco8:
98     pushg 64
99     pushi 0
100    sup
101    jz bloco9
102    pushg 64
103    pushi 1
104    sub
105    storeg 64
106    pushgp
107    pushi 0
108    padd
109    pushg 64
110    loadn
111    writei
112    pushs " \n"
113    writes
114    jump bloco8
115 bloco9:
116 stop

```

Programa que lê N inteiros e imprime o menor

Programas/menorNum.plang

```

1 VAR:
2   x;
3   y;
4   z;
5   w;
6 START:
7   Pprint("Quantos inteiros?\n");
8   Pread(x);
9   z -> 0;
10  if(x > 0) {
11    Pread(y);
12    w- > y;
13    z -> z + 1;
14    while(z < x) {
15      Pread(y);
16      if(y < w) {
17        w -> y;
18      }
19      z -> z + 1;
20    }
21    Pprint("Menor número:");
22    Pprint(w);
23    Pprint(" \n");

```

```

24 }
25 else {
26     Pprint("Não inseriu nenhum número :(\n");
27 }
28 ;

```

Assembly gerado:

Assembly/menorNum.vm

```

1     pushi 0
2     pushi 0
3     pushi 0
4     pushi 0
5 start
6     pushs "Quantos inteiros?\n"
7     writes
8     read
9     atoi
10    storeg 0
11    pushi 0
12    storeg 2
13    pushg 0
14    pushi 0
15    sup
16    jz bloco1
17    read
18    atoi
19    storeg 1
20    pushg 1
21    storeg 3
22    pushg 2
23    pushi 1
24    add
25    storeg 2
26 bloco2:
27    pushg 2
28    pushg 0
29    inf
30    jz bloco3
31    read
32    atoi
33    storeg 1
34    pushg 1
35    pushg 3
36    inf
37    jz bloco4
38    pushg 1
39    storeg 3
40 bloco4:
41    pushg 2
42    pushi 1
43    add
44    storeg 2
45    jump bloco2
46 bloco3:
47    pushs "Menor número:"
48    writes
49    pushg 3

```

```

50     writei
51     pushs " \n"
52     writes
53     jump bloco5
54 bloco1:
55     pushs "Não inseriu nenhum número :(\n"
56     writes
57 bloco5:
58 stop

```

Programa que calcula o produtório de 10 números

Programas/produtorio.plang

```

1  # Programa que calcula o produtório de 10 números
2  VAR:
3      total;
4      produtorio;
5      aux;
6  START:
7      total -> 10;
8      produtorio -> 1;
9      Pprint("Insira 10 números \n");
10     while(total > 0) {
11         Pread(aux);
12         produtorio -> produtorio * aux;
13         total -> total - 1;
14     }
15     Pprint("Produtório: ");
16     Pprint(produtorio);
17     Pprint(" \n");
18 ;

```

Assembly gerado:

Assembly/produtorio.vm

```

1      pushi 0
2      pushi 0
3      pushi 0
4  start
5      pushi 10
6      storeg 0
7      pushi 1
8      storeg 1
9      pushs "Insira 10 números \n"
10     writes
11 bloco1:
12     pushg 0
13     pushi 0
14     sup
15     jz bloco2
16     read
17     atoi
18     storeg 2
19     pushg 1
20     pushg 2
21     mul
22     storeg 1

```



```

23     pushg 0
24     pushi 1
25     sub
26     storeg 0
27     jump bloco1
28 bloco2:
29     pushs "Produtório: "
30     writes
31     pushg 1
32     writei
33     pushs " \n"
34     writes
35 stop

```

Programa que recebe 4 inteiros e diz se são lados de um quadrado

Programas/quadrado.plang

```

1  # Programa que recebe 4 inteiros e diz se são lados de um quadrado
2  VAR:
3      numeros;
4      ultimo;
5      lado;
6      verdade;
7      atual;
8  START:
9      Pprint("Insira 4 números referentes a lados de um quadrado: \n");
10     numeros -> 4;
11     Pread(atual);
12     lado -> atual;
13     verdade -> 1;
14     numeros -> numeros - 1;
15     while(numeros > 0) {
16         Pread(atual);
17         if(atual == lado) {
18             numeros -> numeros - 1;
19         }
20         else {
21             verdade -> 0;
22             numeros -> numeros - 1;
23         }
24     }
25     if(verdade == 1) {
26         Pprint("É um quadrado! \n");
27     }
28     else {
29         Pprint("Não é um quadrado! \n");
30     }
31 ;

```

Assembly gerado:

Assembly/quadrado.vm

```

1     pushi 0
2     pushi 0
3     pushi 0
4     pushi 0
5     pushi 0

```

```

6  start
7      pushs "Insira 4 números referentes a lados de um quadrado: \n"
8      writes
9      pushi 4
10     storeg 0
11     read
12     atoi
13     storeg 4
14     pushg 4
15     storeg 2
16     pushi 1
17     storeg 3
18     pushg 0
19     pushi 1
20     sub
21     storeg 0
22  bloco1:
23     pushg 0
24     pushi 0
25     sup
26     jz bloco2
27     read
28     atoi
29     storeg 4
30     pushg 4
31     pushg 2
32     equal
33     jz bloco3
34     pushg 0
35     pushi 1
36     sub
37     storeg 0
38     jump bloco4
39  bloco3:
40     pushi 0
41     storeg 3
42     pushg 0
43     pushi 1
44     sub
45     storeg 0
46  bloco4:
47     jump bloco1
48  bloco2:
49     pushg 3
50     pushi 1
51     equal
52     jz bloco5
53     pushs "É um quadrado! \n"
54     writes
55     jump bloco6
56  bloco5:
57     pushs "Não é um quadrado! \n"
58     writes
59  bloco6:
60  stop

```

Programa que le e armazena N inteiros num array e imprime em ordem inversa

Programas/inversa.plang

```

1  # Programa que le e armazena N inteiros num array e imprime em ordem inversa
2  VAR:
3      numeros[64];
4      quantos;
5      indice;
6  START:
7      Pprint("Quantos inteiros?\n");
8      Pread(quantos);
9      indice -> 0;
10
11     while(indice < quantos) {
12         Pread(numeros[indice]);
13         indice -> indice + 1;
14     }
15
16     Pprint("Array em ordem inversa: \n");
17
18     while(quantos > 0) {
19         quantos -> quantos - 1;
20         Pprint(numeros[quantos]);
21         Pprint(" \n");
22     }
23 ;

```

Assembly gerado:

Assembly/inversa.vm

```

1      pushn 64
2      pushi 0
3      pushi 0
4  start
5      pushs "Quantos inteiros?\n"
6      writes
7      read
8      atoi
9      storeg 64
10     pushi 0
11     storeg 65
12  bloco1:
13     pushg 65
14     pushg 64
15     inf
16     jz bloco2
17     pushgp
18     pushi 0
19     padd
20     pushg 65
21     read
22     atoi
23     storen
24     pushg 65
25     pushi 1
26     add
27     storeg 65
28     jump bloco1
29  bloco2:
30     pushs "Array em ordem inversa: \n"

```

```

31     writes
32 bloco3:
33     pushg 64
34     pushi 0
35     sup
36     jz bloco4
37     pushg 64
38     pushi 1
39     sub
40     storeg 64
41     pushgp
42     pushi 0
43     padd
44     pushg 64
45     loadn
46     writei
47     pushs " \n"
48     writes
49     jump bloco3
50 bloco4:
51 stop

```

Programa - Contar e imprimir os numeros impares de uma sequencia de numeros naturais(pára no 0)

Programas/impares.plang

```

1  # //Programa// - //contar e imprimir os numeros impares de uma sequencia de numeros
   naturais(pára no 0)//
2  VAR:
3      lido;
4      quantos;
5  START:
6      quantos -> 0;
7
8      Pprint("Insira numeros naturais(0 para terminar)\n");
9      Pread(lido);
10
11     while(lido > 0) {
12         if(lido % 2 == 0) {
13             Pread(lido);
14         }
15         else {
16             quantos -> quantos + 1;
17             Pprint("Ímpar: ");
18             Pprint(lido);
19             Pprint(" \n");
20             Pread(lido);
21         }
22     }
23     Pprint("Foram lidos: ");
24     Pprint(quantos);
25     Pprint(" impares.\n");
26 ;

```

Assembly gerado:

Assembly/impares.vm

```

1      pushi 0
2      pushi 0
3  start
4      pushi 0
5      storeg 1
6      pushs "Insira numeros naturais(0 para terminar)\n"
7      writes
8      read
9      atoi
10     storeg 0
11  bloco1:
12     pushg 0
13     pushi 0
14     sup
15     jz bloco2
16     pushg 0
17     pushi 2
18     mod
19     pushi 0
20     equal
21     jz bloco3
22     read
23     atoi
24     storeg 0
25     jump bloco4
26  bloco3:
27     pushg 1
28     pushi 1
29     add
30     storeg 1
31     pushs "Ímpar: "
32     writes
33     pushg 0
34     writei
35     pushs " \n"
36     writes
37     read
38     atoi
39     storeg 0
40  bloco4:
41     jump bloco1
42  bloco2:
43     pushs "Foram lidos: "
44     writes
45     pushg 1
46     writei
47     pushs " impares.\n"
48     writes
49  stop

```

Capítulo 5

Conclusão

Neste projeto foi possível atingir os objetivos propostos, relativos à criação de um compilador baseado numa gramática tradutora. Para além disso, foi possível conciliar os conceitos relativos a gramáticas independentes do contexto, capazes de usar BNF-puro e satisfazer a condição LR().

A principal dificuldade passou construir uma gramática suficientemente e completa, sem quaisquer ambiguidades e incapaz de criar conflitos. Apesar disso, na nossa opinião, a gramática criada encontra-se bastante razoável e capaz de caracterizar de um modo completo a linguagem desenvolvida.

De um modo geral, fazemos uma avaliação positiva do trabalho. Todos os requisitos base expressos no enunciado foram satisfeitos e algumas funcionalidades extra adicionadas. Em relação às funcionalidades extra, no presente trabalho foram realizadas em menor número, em comparação com os trabalhos anteriores, muito devido ao fator tempo.

Apêndice A

Código do Programa

Lista-se a seguir o código do programa que foi desenvolvido.

plang.flex

```
1 %option noyywrap yylineno
2
3
4 %%
5 -?[0-9]+      {
6               yynlval.d = atoi(yytext);
7               return INTGR;
8               }
9 [\^+\-*/\[\]=<>,(%)%:{\}\|&! ] {
10              return yytext[0];
11              }
12 VAR           {
13              return VAR;
14              }
15 START         {
16              return START;
17              }
18 Pread         {
19              return PREAD;
20              }
21 Pprint        {
22              return PPRINT;
23              }
24 if            {
25              return IF;
26              }
27 else          {
28              return ELSE;
29              }
30 while         {
31              return WHILE;
32              }
33 [a-z]+        {
34              yynlval.s = strdup(yytext);
35              return V;
36              }
37 \"[\^\\n]*\"    {
```

```

38         yyval.s = strdup(yytext);
39         return STRING;
40     }
41     #.+\\n          { }
42     [ \\t\\n]      { }
43
44     .|\\n          { yyerror("Caracter desconhecido"); }
45
46     %%

```

plang.y

```

1  %{
2      #include <stdio.h>
3      #include <stdlib.h>
4      #include <math.h>
5      #include <glib.h>
6      #include <string.h>
7      #include <ctype.h>
8
9      #define MAXSTACK 512
10
11     typedef struct variable {
12         int stack;
13         int size;
14     } *Var;
15
16     typedef struct stack {
17         int blocks[MAXSTACK];
18         int top;
19     } *Stack;
20
21     int pop(Stack);
22     void push(int, Stack);
23     void yyerror(char*);
24     int yylex();
25     Var addVar(Var, int, int);
26
27     GHashTable* variaveis;
28     Stack ifs;
29     Stack whiles;
30     FILE *fp;
31     int sp = 0;
32     int blocos = 0;
33
34
35     Var temp;
36     Var v = NULL;
37
38     %}
39     %token VAR V INTGR START PREAD PPRINT IF ELSE WHILE STRING
40
41     %union { char *s; int d; }
42     %type<d> INTGR
43     %type<s> V STRING
44
45     %%
46     Plang : Init Body { fprintf(fp, "stop\\n"); }
47           ;

```



```

48
49 Init      : VAR ':' Declare { fprintf(fp, "start\n"); }
50           ;
51
52 Declare    : Declare Variable
53           | {
54             variaveis = g_hash_table_new(g_str_hash, g_str_equal);
55             ifs = (Stack)malloc(sizeof(struct stack));
56             ifs -> top = 0;
57             whiles = (Stack)malloc(sizeof(struct stack));
58             whiles -> top = 0;
59           }
60           ;
61
62 Variable   : V ';' {
63             temp = g_hash_table_lookup(variaveis,$1);
64             if(temp) {
65                 yyerror("Variável já declarada anteriormente!");
66             }
67             else {
68                 v = addVar(v,sp,1);
69                 g_hash_table_insert(variaveis,$1,v);
70                 fprintf(fp, "    pushi 0\n");
71                 sp++;
72             }
73         }
74         | V '[' INTGR ']' ';' {
75             temp = g_hash_table_lookup(variaveis,$1);
76             if(temp) {
77                 yyerror("Variável já declarada anteriormente!");
78             }
79             else {
80                 v = addVar(v,sp,$3);
81                 g_hash_table_insert(variaveis,$1,v);
82                 fprintf(fp, "    pushn %d\n", $3);
83                 sp += $3;
84             }
85         }
86         | V '{' INTGR '}' '[' INTGR ']' ';' {
87             temp = g_hash_table_lookup(variaveis,$1);
88             if(temp) {
89                 yyerror("Variável já declarada anteriormente!");
90             }
91             else {
92                 v = addVar(v,sp,$6);
93                 g_hash_table_insert(variaveis,$1,v);
94                 fprintf(fp, "    pushn %d\n", $3*$6);
95                 sp += $3*$6;
96             }
97         }
98         ;
99
100 Body       : START ':' Instructions ';'
101           ;
102
103 Instructions: Instructions Instruction
104           | Instruction
105           ;
106

```

```

107 Instruction : Assignment
108     | Read
109     | Print
110     | Condicional
111     | Cyclic
112     ;
113
114 Assignment : V '-' '>' Expression ';' {
115     temp = g_hash_table_lookup(variaveis,$1);
116     if(!temp) {
117         yyerror("A variável não foi anteriormente declarada!");
118     }
119     else {
120         fprintf(fp, "    storeg %d\n", temp->stack);
121     }
122 }
123 | V {
124     temp = g_hash_table_lookup(variaveis,$1);
125     if(!temp) {
126         yyerror("A variável não foi anteriormente declarada!");
127     }
128     else {
129         fprintf(fp, "    pushgp\n");
130         fprintf(fp, "    pushi %d\n", temp->stack);
131         fprintf(fp, "    padd\n");
132     }
133 } '[' Expression ']' '-' '>' Expression ';' {
134     fprintf(fp, "    storen\n");
135 }
136 | V '{' {
137     temp = g_hash_table_lookup(variaveis,$1);
138     if(!temp) {
139         yyerror("A variável não foi anteriormente declarada!");
140     }
141     else {
142         fprintf(fp, "    pushgp\n");
143         fprintf(fp, "    pushi %d\n", temp->stack);
144         fprintf(fp, "    padd\n");
145     }
146 } Expression '}' '[' {
147     fprintf(fp, "    pushi %d\n", temp->size);
148     fprintf(fp, "    mul\n");
149 } Expression ']' '-' '>' {
150     fprintf(fp, "    add\n");
151 } Expression ';' {
152     fprintf(fp, "    storen\n");
153 }
154 ;
155
156 Read : PREAD '(' V ')' ';' {
157     temp = g_hash_table_lookup(variaveis,$3);
158     if(temp == NULL) {
159         yyerror("A variável não foi anteriormente declarada!");
160     }
161     else {
162         fprintf(fp, "    read\n");
163         fprintf(fp, "    atoi\n");
164         fprintf(fp, "    storeg %d\n", temp->stack);
165     }

```

```

166 }
167 | PREAD '(' V {
168     temp = g_hash_table_lookup(variaveis,$3);
169     if(!temp) {
170         yyerror("A variável não foi anteriormente declarada!");
171     }
172     else {
173         fprintf(fp, "    pushgp\n");
174         fprintf(fp, "    pushi %d\n", temp->stack);
175         fprintf(fp, "    padd\n");
176     }
177 } '[' Expression ']' ';' {
178     fprintf(fp, "    read\n");
179     fprintf(fp, "    atoi\n");
180     fprintf(fp, "    storen\n");
181 }
182 | PREAD '(' V '{' {
183     temp = g_hash_table_lookup(variaveis,$3);
184     if(!temp) {
185         yyerror("A variável não foi anteriormente declarada!");
186     }
187     else {
188         fprintf(fp, "    pushgp\n");
189         fprintf(fp, "    pushi %d\n", temp->stack);
190         fprintf(fp, "    padd\n");
191     }
192 } Expression '}' '[' {
193     fprintf(fp, "    pushi %d\n", temp->size);
194     fprintf(fp, "    mul\n");
195 } Expression ']' ')' ';' {
196     fprintf(fp, "    add\n");
197     fprintf(fp, "    read\n");
198     fprintf(fp, "    atoi\n");
199     fprintf(fp, "    storen\n");
200 }
201 ;
202
203
204 Print : PPRINT '(' Expression ')' ';' {
205     fprintf(fp, "    writei\n");
206 }
207 | PPRINT '(' STRING ')' ';' {
208     fprintf(fp, "    pushs %s\n", $3);
209     fprintf(fp, "    writes\n");
210 }
211 ;
212
213 Condicional : IF '(' Accumulator ')' {
214     blocos++;
215     fprintf(fp, "    jz bloco%d\n", blocos);
216     push(blocos, ifs);
217 } '{' Instructions '}' Opelse
218 ;
219
220 Opelse : ELSE {
221     blocos++;
222     fprintf(fp, "    jump bloco%d\n", blocos);
223     fprintf(fp, "bloco%d:\n", pop(ifs));
224     push(blocos, ifs);

```

```

225     } '{' Instructions '}' {
226     fprintf(fp, "bloco%d:\n", pop(ifs));
227     }
228     | {
229     fprintf(fp, "bloco%d:\n", pop(ifs));
230     }
231     ;
232
233 Cyclic      : WHILE {
234     blocos++;
235     fprintf(fp, "bloco%d:\n", blocos);
236     push(blocos, whiles);
237     } '(' Accumulator ')' {
238     blocos++;
239     fprintf(fp, "    jz bloco%d\n", blocos);
240     push(blocos, ifs);
241     } '{' Instructions '}' {
242     fprintf(fp, "    jump bloco%d\n", pop(whiles));
243     fprintf(fp, "bloco%d:\n", pop(ifs));
244     }
245     ;
246
247 Accumulator : Comparator '|' '|' Accumulator
248             | Comparator '&' '&' Accumulator
249             | Comparator
250             ;
251
252 Comparator  : Expression
253             | Expression '=' '=' Expression {
254             fprintf(fp, "    equal\n");
255             }
256             | Expression '!' '=' Expression {
257             fprintf(fp, "    equal\n");
258             fprintf(fp, "    not\n");
259             }
260             | Expression '>' Expression {
261             fprintf(fp, "    sup\n");
262             }
263             | Expression '<' Expression {
264             fprintf(fp, "    inf\n");
265             }
266             | Expression '>' '=' Expression {
267             fprintf(fp, "    supeq\n");
268             }
269             | Expression '<' '=' Expression {
270             fprintf(fp, "    ineq\n");
271             }
272             ;
273
274 Expression  : Expression '+' P {
275             fprintf(fp, "    add\n");
276             }
277             | Expression '-' P {
278             fprintf(fp, "    sub\n");
279             }
280             | P
281             ;
282
283 P           : P '*' Fat {

```

```

284     fprintf(fp, "    mul\n");
285 }
286 | P '/' Fat {
287     fprintf(fp, "    div\n");
288 }
289 | P '%' Fat {
290     fprintf(fp, "    mod\n");
291 }
292 | Fat
293 ;
294
295 Fat    : Es '^' Fat
296 | Es
297 ;
298
299 Es     : '(' Expression ')'
300 | INTGR {
301     fprintf(fp, "    pushi %d\n", $1);
302 }
303 | '-' INTGR {
304     fprintf(fp, "    pushi -%d\n", $2);
305 }
306 | V {
307     temp = g_hash_table_lookup(variaveis,$1);
308     if(!temp) {
309         yyerror("A variável não foi anteriormente declarada!");
310     }
311     else {
312         fprintf(fp, "    pushg %d\n", temp->stack);
313     }
314 }
315 | '-' V {
316     temp = g_hash_table_lookup(variaveis,$2);
317     if(!temp) {
318         yyerror("A variável não foi anteriormente declarada!");
319     }
320     else {
321         fprintf(fp, "    pushi -1\n");
322         fprintf(fp, "    pushg %d\n", temp->stack);
323         fprintf(fp, "    mul\n");
324     }
325 }
326 | V {
327     temp = g_hash_table_lookup(variaveis,$1);
328     if(!temp) {
329         yyerror("A variável não foi anteriormente declarada!");
330     }
331     else {
332         fprintf(fp, "    pushgp\n");
333         fprintf(fp, "    pushi %d\n", temp->stack);
334         fprintf(fp, "    padd\n");
335     }
336 } '[' Expression ']' {
337     fprintf(fp, "    loadn\n");
338 }
339 | '-' V {
340     temp = g_hash_table_lookup(variaveis,$2);
341     if(!temp) {
342         yyerror("A variável não foi anteriormente declarada!");

```

```

343     }
344     else {
345         fprintf(fp, "    pushgp\n");
346         fprintf(fp, "    pushi %d\n", temp->stack);
347         fprintf(fp, "    padd\n");
348     }
349 } '[' Expression ']' {
350     fprintf(fp, "    loadn\n");
351     fprintf(fp, "    pushi -1\n");
352     fprintf(fp, "    mul\n");
353 }
354 | V '{' {
355     temp = g_hash_table_lookup(variaveis,$1);
356     if(!temp) {
357         yyerror("A variável não foi anteriormente declarada!");
358     }
359     else {
360         fprintf(fp, "    pushgp\n");
361         fprintf(fp, "    pushi %d\n", temp->stack);
362         fprintf(fp, "    padd\n");
363     }
364 } Expression '}' '[' {
365     fprintf(fp, "    pushi %d\n", temp->size);
366     fprintf(fp, "    mul\n");
367 } Expression ']' {
368     fprintf(fp, "    add\n");
369     fprintf(fp, "    loadn\n");
370 }
371 | '-' V '{' {
372     temp = g_hash_table_lookup(variaveis,$2);
373     if(!temp) {
374         yyerror("A variável não foi anteriormente declarada!");
375     }
376     else {
377         fprintf(fp, "    pushgp\n");
378         fprintf(fp, "    pushi %d\n", temp->stack);
379         fprintf(fp, "    padd\n");
380     }
381 } Expression '}' '[' {
382     fprintf(fp, "    pushi %d\n", temp->size);
383     fprintf(fp, "    mul\n");
384 } Expression ']' {
385     fprintf(fp, "    add\n");
386     fprintf(fp, "    loadn\n");
387     fprintf(fp, "    pushi -1\n");
388     fprintf(fp, "    mul\n");
389 }
390 ;
391 %%
392
393 #include "lex.yy.c"
394
395 void yyerror(char *s) {
396     fprintf(stderr, "linha %d: [%s] - %s\n", yylineno, yytext, s);
397 }
398
399 Var addVar(Var v, int index, int size) {
400     v = (Var) malloc(sizeof(struct variable));
401     v->stack = index;

```

```

402     v->size = size;
403     return v;
404 }
405
406 void push(int i, Stack s) {
407     (s -> blocks)[s -> top] = i;
408     (s -> top)++;
409 }
410
411 int pop(Stack s) {
412     (s -> top)--;
413     int res = (s -> blocks)[s -> top];
414     return res;
415 }
416
417 int main(int argc, char **argv){
418     if(argc > 1) {
419         yyin = fopen(argv[1], "r");
420         if(argc > 2)
421             fp = fopen(argv[2], "w");
422     }
423     else
424         fp = fopen("out.vm", "w");
425     yyparse();
426     return 0;
427 }

```