Compilador para VM usando Yacc e Flex

Processamento de Linguagens (3º ano de Curso)

Trabalho Prático 2

Relatório de Desenvolvimento

José Silva	Pedro Cunha	Gonçalo Moreira
(A74601)	(A73958)	(A73591)

12 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

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 sumarizada 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, atómicas 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 inicio do programa. A primeira declaração deve ser sempre antecedida por "VAR:". Qualquer declaração deve ser precedida por ";". É possível declarar variáveis atómicas mas igualmente variáveis estruturadas. No que diz respeito às ultimas, 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 inicio 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 -¿"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 atómica 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

```
G = <T,N,S,P>
T = {VAR,':',';','[',']','{', '}', START, '-', '>', '(', ')', V,
PREAD, PPRINT, STRING, IF, ELSE, WHILE, '|', '&', '=', '!', '<', '+',
'-', '*', '/', '%', '^', INTGR }
N = {Plang,Init,Body,Declare,Variable,Body,Instruction, Instructions,
Assignment, Read, Print, Condicional, Opelse, Cyclic, Accumulator, Comparator,
Expression, P, Fat, Es}
S = {Plang}

p0: Plang       ->  Init Body
p1: Init        ->  VAR ':' Declare
p2: Declare       ->  Declare Variable
p3: Variable      ->  V ';'
p4:           |  V '[' INTGR ']' ';'
p5:           |  V '{' INTGR '}' '[' INTGR ']' ';'

p6: Body        ->  START ':' Instructions ';'
P7: Instructions    ->  Instructions Instruction
p8:           |   Instruction

p9: Instruction     ->  Assignment
p10:          |  Read
p11:          |  Print
p12:          |  Condicional
p13:          |  Cyclic

p14: Assignment     ->  V '-' '>' Expression ';'
p15:          |   V '[' Expression ']' '-' '>' Expression ';'
p16:          |   V '{' Expression '}' '[' Expression ']' '-' '>' Expression ';'

p17: Read       ->  PREAD '(' V ')' ';'
p18:          |  PREAD '(' V  { '[' Expression ']' ')' ';'
p19:          |  PREAD '(' V '{' { Expression '}' '[' Expression ']' ')' ';'

p20: Print        ->  PPRINT '(' Expression ')' ';'
p21:          |  PPRINT '(' STRING ')' ';'

```

```
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 ultimo 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 inicio 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 inicio 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 (inicio 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*(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 especifico 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 variavel global(inteiro) que garante que nao 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 tambem 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 inicio 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 especificas 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 auxilio 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 consegui 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
```

```
  i -> 0;
  while(i < quantos) {
    Pread(numeros[i]);
    i -> i + 1;
  }

  i -> 0;
  while(i < quantos){
    j -> i + 1;
    while(j < quantos){
      if(numeros[i] > numeros[j]) {
        aux -> numeros[i];
        numeros[i] -> numeros[j];
        numeros[j] -> aux;
      }
      j -> j + 1;
    }
    i -> i + 1;
  }

  Pprint("Array em ordem decrescente: \n");

  while(quantos > 0) {
    quantos -> quantos - 1;
    Pprint(numeros[quantos]);
    Pprint(" \n");
  }
;
```

Assembly gerado:

Assembly/ordena.vm

```
    pushn 64
    pushi 0
    pushi 0
    pushi 0
    pushi 0
start
    pushs "Quantos inteiros?(Max:64)\n"
    writes
    read
    atoi
    storeg 64
    pushi 0
    storeg 66
bloco1:
    pushg 66
    pushg 64
    inf
    jz bloco2
    pushgp
    pushi 0
    padd
    pushg 66
    read
    atoi
    storen
    pushg 66
```

```
    pushi 1
    add
    storeg 66
    jump bloco1
bloco2:
    pushi 0
    storeg 66
bloco3:
    pushg 66
    pushg 64
    inf
    jz bloco4
    pushg 66
    pushi 1
    add
    storeg 67
bloco5:
    pushg 67
    pushg 64
    inf
    jz bloco6
    pushgp
    pushi 0
    padd
    pushg 66
    loadn
    pushgp
    pushi 0
    padd
    pushg 67
    loadn
    sup
    jz bloco7
    pushgp
    pushi 0
    padd
    pushg 66
    loadn
    storeg 65
    pushgp
    pushi 0
    padd
    pushg 66
    pushgp
    pushi 0
    padd
    pushg 67
    loadn
    storen
    pushgp
    pushi 0
    padd
    pushg 67
    pushg 65
    storen
bloco7:
    pushg 67
    pushi 1
    add
```

```
    storeg 67
    jump bloco5
bloco6:
    pushg 66
    pushi 1
    add
    storeg 66
    jump bloco3
bloco4:
    pushs "Array em ordem decrescente: \n"
    writes
bloco8:
    pushg 64
    pushi 0
    sup
    jz bloco9
    pushg 64
    pushi 1
    sub
    storeg 64
    pushgp
    pushi 0
    padd
    pushg 64
    loadn
    writei
    pushs " \n"
    writes
    jump bloco8
bloco9:
stop
```

Programa que lê N inteiros e imprime o menor

Programas/menorNum.plang

```
VAR:
  x;
  y;
  z;
  w;
START:
  Pprint("Quantos inteiros?\n");
  Pread(x);
  z -> 0;
  if(x > 0) {
    Pread(y);
    w- > y;
    z -> z + 1;
    while(z < x) {
      Pread(y);
      if(y < w) {
        w -> y;
      }
      z -> z + 1;
    }
    Pprint("Menor número:");
    Pprint(w);
    Pprint(" \n");
```

```
  }
  else {
    Pprint("Não inseriu nenhum número :(\n");
  }
;
```

Assembly gerado:

Assembly/menorNum.vm

```
    pushi 0
    pushi 0
    pushi 0
    pushi 0
start
    pushs "Quantos inteiros?\n"
    writes
    read
    atoi
    storeg 0
    pushi 0
    storeg 2
    pushg 0
    pushi 0
    sup
    jz bloco1
    read
    atoi
    storeg 1
    pushg 1
    storeg 3
    pushg 2
    pushi 1
    add
    storeg 2
bloco2:
    pushg 2
    pushg 0
    inf
    jz bloco3
    read
    atoi
    storeg 1
    pushg 1
    pushg 3
    inf
    jz bloco4
    pushg 1
    storeg 3
bloco4:
    pushg 2
    pushi 1
    add
    storeg 2
    jump bloco2
bloco3:
    pushs "Menor número:"
    writes
    pushg 3
```

```
    writei
    pushs " \n"
    writes
    jump bloco5
bloco1:
    pushs "Não inseriu nenhum número :(\n"
    writes
bloco5:
stop
```

Programa que calcula o produtório de 10 números

Programas/produtorio.plang

```
# Programa que calcula o produtório de 10 números
VAR:
  total;
  produtorio;
  aux;
START:
  total -> 10;
  produtorio -> 1;
  Pprint("Insira 10 números \n");
  while(total > 0) {
    Pread(aux);
    produtorio -> produtorio * aux;
    total -> total - 1;
  }
  Pprint("Produtório: ");
  Pprint(produtorio);
  Pprint(" \n");
;
```

Assembly gerado:

Assembly/produtorio.vm

```
    pushi 0
    pushi 0
    pushi 0
start
    pushi 10
    storeg 0
    pushi 1
    storeg 1
    pushs "Insira 10 números \n"
    writes
bloco1:
    pushg 0
    pushi 0
    sup
    jz bloco2
    read
    atoi
    storeg 2
    pushg 1
    pushg 2
    mul
    storeg 1
```

```
    pushg 0
    pushi 1
    sub
    storeg 0
    jump bloco1
bloco2:
    pushs "Produtório: "
    writes
    pushg 1
    writei
    pushs " \n"
    writes
stop
```

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

Programas/quadrado.plang

```
# Programa que recebe 4 inteiros e diz se sao lados de um quadrado
VAR:
  numeros;
  ultimo;
  lado;
  verdade;
  atual;
START:
  Pprint("Insira 4 números referentes a lados de um quadrado: \n");
  numeros -> 4;
  Pread(atual);
  lado -> atual;
  verdade -> 1;
  numeros -> numeros - 1;
  while(numeros > 0) {
    Pread(atual);
    if(atual == lado) {
      numeros -> numeros - 1;
    }
    else {
      verdade -> 0;
      numeros -> numeros - 1;
    }
  }
  if(verdade == 1) {
    Pprint("É um quadrado! \n");
  }
  else {
    Pprint("Não é um quadrado! \n");
  }
;
```

Assembly gerado:

Assembly/quadrado.vm

```
    pushi 0
    pushi 0
    pushi 0
    pushi 0
    pushi 0
```

```
start
    pushs "Insira 4 números referentes a lados de um quadrado: \n"
    writes
    pushi 4
    storeg 0
    read
    atoi
    storeg 4
    pushg 4
    storeg 2
    pushi 1
    storeg 3
    pushg 0
    pushi 1
    sub
    storeg 0
bloco1:
    pushg 0
    pushi 0
    sup
    jz bloco2
    read
    atoi
    storeg 4
    pushg 4
    pushg 2
    equal
    jz bloco3
    pushg 0
    pushi 1
    sub
    storeg 0
    jump bloco4
bloco3:
    pushi 0
    storeg 3
    pushg 0
    pushi 1
    sub
    storeg 0
bloco4:
    jump bloco1
bloco2:
    pushg 3
    pushi 1
    equal
    jz bloco5
    pushs "É um quadrado! \n"
    writes
    jump bloco6
bloco5:
    pushs "Não é um quadrado! \n"
    writes
bloco6:
stop
```

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

Programas/inversa.plang

```
# Programa que le e armazena N inteiros num array e imprime em ordem inversa
VAR:
  numeros[64];
  quantos;
  indice;
START:
  Pprint("Quantos inteiros?\n");
  Pread(quantos);
  indice -> 0;

  while(indice < quantos) {
    Pread(numeros[indice]);
    indice -> indice + 1;
  }

  Pprint("Array em ordem inversa: \n");

  while(quantos > 0) {
    quantos -> quantos - 1;
    Pprint(numeros[quantos]);
    Pprint(" \n");
  }
;
```

Assembly gerado:

Assembly/inversa.vm

```
    pushn 64
    pushi 0
    pushi 0
start
    pushs "Quantos inteiros?\n"
    writes
    read
    atoi
    storeg 64
    pushi 0
    storeg 65
bloco1:
    pushg 65
    pushg 64
    inf
    jz bloco2
    pushgp
    pushi 0
    padd
    pushg 65
    read
    atoi
    storen
    pushg 65
    pushi 1
    add
    storeg 65
    jump bloco1
bloco2:
    pushs "Array em ordem inversa: \n"
```

```
    writes
bloco3:
    pushg 64
    pushi 0
    sup
    jz bloco4
    pushg 64
    pushi 1
    sub
    storeg 64
    pushgp
    pushi 0
    padd
    pushg 64
    loadn
    writei
    pushs " \n"
    writes
    jump bloco3
bloco4:
stop
```

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

Programas/impares.plang

```
# //Programa// - //contar e imprimir os numeros impares de uma sequencia de numeros
    naturais(pára no 0)//
VAR:
  lido;
  quantos;
START:
  quantos -> 0;

  Pprint("Insira numeros naturais(0 para terminar)\n");
  Pread(lido);

  while(lido > 0) {
    if(lido % 2 == 0) {
      Pread(lido);
    }
    else {
      quantos -> quantos + 1;
      Pprint("Ímpar: ");
      Pprint(lido);
      Pprint(" \n");
      Pread(lido);
    }
  }
  Pprint("Foram lidos: ");
  Pprint(quantos);
  Pprint(" impares.\n");
;
```

Assembly gerado:

Assembly/impares.vm

```
    pushi 0
    pushi 0
start
    pushi 0
    storeg 1
    pushs "Insira numeros naturais(0 para terminar)\n"
    writes
    read
    atoi
    storeg 0
bloco1:
    pushg 0
    pushi 0
    sup
    jz bloco2
    pushg 0
    pushi 2
    mod
    pushi 0
    equal
    jz bloco3
    read
    atoi
    storeg 0
    jump bloco4
bloco3:
    pushg 1
    pushi 1
    add
    storeg 1
    pushs "Ímpar: "
    writes
    pushg 0
    writei
    pushs " \n"
    writes
    read
    atoi
    storeg 0
bloco4:
    jump bloco1
bloco2:
    pushs "Foram lidos: "
    writes
    pushg 1
    writei
    pushs " impares.\n"
    writes
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

```
%option noyywrap yylineno


%%
-?[0-9]+        {
               yylval.d = atoi(yytext);
               return INTGR;
               }
[\^+\-*/\[\]=><;,()%:{}\|&!]    {
               return yytext[0];
               }
VAR                            {
               return VAR;
               }
START                          {
               return START;
               }
Pread                          {
               return PREAD;
               }
Pprint                         {
               return PPRINT;
               }
if                             {
               return IF;
               }
else                           {
               return ELSE;
               }
while                          {
               return WHILE;
               }
[a-z]+                         {
               yylval.s = strdup(yytext);
               return V;
               }
\"[^\n]*\"     {
```

```
38              yylval.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, "   infeq\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 }
```