Processamento de Linguagens - Fase 2

Gonçalo Medeiros, Luís Magalhães, Rúben Lucas

May 30, 2021

A89514
Gonçalo Medeiros

A89487
Rúben Lucas

A89528
Luís Magalhães

Contents

1 Introdução

O intuito deste trabalho é aplicar os conhecimentos aprendidos sobre gramáticas na disciplina de Processamento de Linguagens.

1.1 Objectivos

- aumentar a experiência em engenharia de linguagens e em programação generativa aprimorando a capacidade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT);
- desenvolver processadores de linguagens a partir de uma gramática tradutora;
- desenvolver um compilador gerando código para uma máquina de stack virtual (VM, Virtual Machine1)
- utilizar geradores de compiladores baseados em gramáticas tradutoras (*Yac* e *Lex*, versões *PLY* do Python).

1.2 Enunciado

Pretende-se que comece por definir uma linguagem de programação imperativa simples, a seu gosto.
Apenas deve ter em consideração que essa linguagem terá de permitir:

- *declarar* variáveis atómicas do tipo *inteiro*, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas.
- *efetuar* instruções algorítmicas básicas como a *atribuição do valor de expressões numéricas a variáveis.*
- *ler* do *standard input* e *escrever* no *standard output.*
- *efetuar* instruções *condicionais* para controlo do fluxo de execução.
- *efetuar* instruções *cíclicas* para controlo do fluxo de execução, permitindo o seu aninhamento.
  Note que deve implementar pelo menos o ciclo `while-do`, `repeat-until` ou `for-do` conforme o Número do seu Grupo módulo 3 seja 0, 1 ou 2.

Adicionalmente deve ainda suportar, à sua escolha, uma das duas funcionalidades seguintes:

- *declarar e manusear* variáveis estruturadas do tipo *array (a 1 ou 2 dimensões) de inteiros*, em relação aos quais é apenas permitida a operação de indexação (índice inteiro).
- *definir e invocar subprogramas* sem parâmetros mas que possam retornar um resultado do tipo inteiro.

Figure 1: Enunciado

1.3 Descrição do problema

Como já referido anteriormente, o nosso problema consiste em definir a nossa própria linguagem de programação imperativa, tendo que ter em conta as especificações indicadas pelo enunciado. Para este fim, é-nos pedido para desenvolver um compilador gerando código para uma máquina virtual e utilizar o gerador de compilador baseado em gramáticas tradutoras, nomeadamente o *Yac* e *Lex*, versões *PLY* do Python.

————- das decis˜oes que lideraram o desenho da linguagem/gram´atica e as regras de tradu¸c˜ao para Assembly da VM (incluir as especifica¸c˜oes Yacc), dever´a conter exemplos de utiliza¸c˜ao (programas-fonte diversos e respectivo c´odigo produzido). ————-

2 Estrutura do projecto

2.1 Descrição da linguagem

Como indicado, primeiramente temos que proceder à declaração de variáveis e estas não podem ser redeclaradas posteriormente, nem podem ser utilizadas

sem declaração. Se nenhum valor for atribuído a uma variável, esta assumirá o valor zero. É de realçar que temos dois tipos de variáveis, inteiros e *arrays* de inteiros.
Após a declaração de todas as variáveis a serem utilizadas ao longo do programa, segue-se uma *keyword start* que indicará o início do código. Como tal, após esta palavra reservada estará toda a informação pertinente ao funcionamento do programa propriamente dito.

A nossa linguagem suporta os seguintes tipos de instruções:

- Instruções algorítmicas como a atribuição de valores numéricos a variáveis
  -Para alcançar este fim iguala-se o nome da variável ao valor desejado.
- Instruções cíclicas para controlo de fluxo, com aninhamento
  -Implementámos os ciclos *while* e *repeat*, no primeiro é dada uma condição e o código é executado até esta condição retornar falso/zero e no segundo é dado um inteiro N e o ciclo executará N vezes. Apenas o ciclo *while* permite aninhamentos.
- Instruções condicionais para controlo de fluxo de execução.
  -Suporte das estruturas *if else* com encadeamento das mesmas.
- Ler do *stdin* e escrever no *stdout*
  - Por fim a nossa linguagem permite estas funcionalidades através do uso das *keywords print*, *printS* e *read* que permitem a escrita no *stdout* de variáveis/inteiros, escrita no *stdout* de *Strings* e a leitura do *stdin*, respetivamente.

2.2 Estados do *Parser*

Começámos por criar as seguintes variáveis relativas ao *parser*

```
1 # parser state
2 parser.var = {}
3 parser.id = 0
4 parser.startup = len(parser.var)
```

O *var* representa o dicionário onde é guardada a informação essencial sobre as variáveis declaradas no programa (i.e o seu nome, a sua posição e, no caso de se tratar de um *array*, o seu tamanho).
A variável *id* é utilizado como forma de não repetir o nome das *tags* de *jump*

O *startup* representa a quantidade de *push* 0 que existem na declaração de variáveis.

2.3 Regras de Produção

2.3.1 Início

Rule 0 S' → Program
Rule 1 Program → Start Code
Rule 2 Start → Declaration start

Produção que divide o programa na secção de declaração de variáveis iniciais, antes de encontrar a *keyword start*, do resto do programa.

2.3.2 Declaration

Rule 3 Declaration → int Attr Declaration
Rule 4 Declaration → int id [integer] ; Declaration
Rule 5 Declaration → empty
*Rule*6 *Attr*→ id , Attr
Rule 7 Attr → id = Atomic , Attr
Rule 8 Attr → id = integer
Rule 9 Attr → id ;

Produção que separa os vários tipos possíveis de declaração de variáveis.

2.3.3 Code

Rule 10 Code → Line Code
Rule 11 Code → empty
*Rule*12 *Line*→ print (Exp)
Rule 13 Line → printS (String)
Rule 14 Line → read (id)
Rule 15 Line → read (id [integer])
Rule 16 Line → read (id [id])
Rule 17 Line → id = Exp ;
Rule 18 Line → Array = Exp ;
Rule 19 Line → id + + ;
Rule 20 Line → id - - ;

Rule 21 Line → if Ifcond CondCode
Rule 22 Line → WhileStart WhileLoop Code
Rule 23 Line → RepeatStart RepeatLoop Code
Rule 24 Line → (Exp)

Aqui estabelecem-se as regras de produção que constituem o código/linhas de código, elementos responsáveis pelas principais funcionalidades do programa.

2.3.4 While

Rule 25 WhileStart → while
Rule 26 WhileLoop → (Cond)
Rule 27 WhileLoop → (error)

2.3.5 Repeat

Rule 28 RepeatStart → repeat (Exp
Rule 29 RepeatStart → repeat (error
Rule 30 RepeatLoop →)

2.3.6 *if else*

Rule 31 Ifcond → (Cond)
Rule 32 Ifcond → (error)
Rule 33 CondCode → IfStart Code }
Rule 34 CondCode → ElseStart Code }
Rule 35 IfStart →
Rule 36 ElseStart → IfStart Code else {

2.3.7 Condicionais

Rule 37 Cond → Cond or CondAnd
Rule 38 Cond → CondAnd
Rule 39 CondAnd → CondAnd and CondNot
Rule 40 CondAnd → CondNot

Rule 41 CondNot → ! Cond !
Rule 42 CondNot → (Cond)
Rule 43 CondNot → Rel
Rule 44 Rel → Exp *Exp*
*Rule*45 *Rel*→ Exp = *Exp*
*Rule*46 *Rel*→ Exp Exp
Rule 47 Rel → Exp = Exp
Rule 48 Rel → Exp = = Exp
Rule 49 Rel → Exp ! = Exp

2.3.8 Array

Rule 50 Array → id [integer]
Rule 51 Array → id [id]

2.3.9 Exp

Rule 52 Exp → Exp + Termo
Rule 53 Exp → Exp - Termo
Rule 54 Exp → Termo
Rule 55 Termo → Termo * Factor
Rule 56 Termo → Termo / Factor
Rule 57 Termo → Termo % Factor
Rule 58 Termo → Factor
Rule 59 Factor → (Exp)
Rule 60 Factor → Atomic
Rule 61 Factor → Signal Atomic
Rule 62 Atomic → integer
Rule 63 Atomic → id
Rule 64 Atomic → id [integer]
Rule 65 Atomic → id [id]
Rule 66 Signal → -
Rule 67 Signal → Signal -

Juntamente com as produções Condicionais são fundamentais para estabelecer prioridades aritméticas do programa

2.3.10 String

Rule 68 String → String + string
Rule 69 String → String + id
Rule 70 String → string
Rule 71 String → id

Produções referentes aos *prints* de strings no standard output

2.4 Tradução para a VM

2.4.1 Line

```
def p_Line_Print_Exp(p):
    "Line : print '(' Exp ')'"
    outputFile.write("\tWRITEI\n")
```

```
def p_Line_Read_Attr(p):
    "Line : read '(' id ')'"
    outputFile.write("\tREAD\n")
    outputFile.write("\tATOI\n")
    outputFile.write("\tSTOREG " + str(p.parser.var.get(p[3])
   ) + "\n")
```

Trata-se do *input* do *stdin*, utilizando as instruções

2.5 Controlo de erros

O nosso compilador efectua controlo de erros de três tipos: erros léxicos, erros semânticos e erros sintácticos. Para todos os erros detectados, o nosso programa escreve uma mensagem descritiva dos mesmos no *stdout*.

2.5.1 Erros léxicos

Quando detectamos um erro léxico, o *token* desconhecido é escrito no *stdout*, bem como a especificação deste e da sua linha. O facto do texto de input ser processado pelo *yacc* dificultou-nos a impressão da coluna correta de modo que a nossa linguagem apenas informa sobre o conteúdo do *token* e da sua linha.

```
def t_error(t):
    print('LEX ERROR:\n\t' + str(t))
```

```
    print("\tIllegal character \'" + t.value[0] + '\'')
    print("\tLine " + str(t.lexer.lineno))
    t.lexer.skip(1)
```

2.5.2 Erros sintácticos

Quando detectamos um erro sintáctico verificamos o tipo de *token* reconhecido. No caso de este se tratar de uma *keyword*, imprimimos um pequeno texto sobre o comportamente expectável desta. No caso de cair no tipo de *token* id (que é o menos restrito de todos), comparamos o conteúdo do *token* com todas as *keywords* e sugerimos a utilização do *token* lexicamente mais semelhante a este. Para além disso, também imprimimos as suas linha e coluna.

```
# Error rule for syntax error
def p_error(p):
    print("YACC ERROR:\n\t" + str(p))
    line_size = p.lexpos - code.rfind('\n', 0, p.lexpos) +1
    print("\tError found on line", p.lineno, "column",
   line_size)
    if p.type == "id":
        if expected := mostSimilar(p.value):
            print("\tYou wrote \'" + p.value + "\', did you
   mean to write \'" + expected + "\'?")
        else:
            print("\tUnkown simbol: " + p.value)
            pass
    elif p.type == "start":
        print("\tKeyword 'start' misused, this keyword is
   utilized in the begining of your code block.")
    elif p.type == "print":
        print("\tKeyword 'print' misused, this keyword prints
    any integer in the standard output.")
    elif p.type == "printS":
        print("\tKeyword 'printS' misused, this keyword
   prints any string of characters in the standard output.")
    elif p.type == "print":
        print("\tKeyword 'read' misused, this keyword reads
   any integer in the standard input.")
    elif p.type == "if":
        print("\tKeyword 'if' misused, this keyword initiates
    an if statement.")
    elif p.type == "else":
```

```
23          print("\tKeyword 'else' misused, this keyword must be
    utilized after an if code block.")
24      elif p.type == "repeat":
25          print("\tKeyword 'repeat' misused, this keyword loops
    the given block of code as many times as the result of
    the given expression.")
26      elif p.type == "while":
27          print("\tKeyword 'while' misused, this keyword loops
    the given block of code as long as the given condition is
    true.")
28      elif p.type == "and":
29          print("\tKeyword 'and' misused, this keyword is the
    logical conjunction opetator.")
30      elif p.type == "or":
31          print("\tKeyword 'or' misused, this keyword is the
    logical disjunction opetator.")
32      elif p.type == "int":
33          print("\tKeyword 'int' misused, this keyword
    indicates the type of a variable.")
34      else:
35          pass
```

2.5.3 Erros semânticos

A nível de erros semânticos, o nosso compilador apenas detecta erros caso haja um uso incorrecto de condicionais ou de variáveis inteiras e *arrays* de inteiros.

```
1  def p_Declaration_Array(p):
2      "Declaration : int id '[' integer ']' ';' Declaration"
3      if id := p.parser.var.get(p[2]):
4          print("Semantic error:")
5          print("\tVariable \'" + p[2] + "\' already defined")
6      else:
7          p.parser.var.update({p[2] : (p.parser.startup, p[4])
       })
8          p.parser.startup += p[4]
9      p[0] = p[7]
10
11 # Production rules for Attr
12 def p_Attr(p):
13     "Attr : id ',' Attr "
14     if type(p.parser.var.get(p[1])):
15         print("Semantic error:")
16         print("\tVariable \'" + p[1] + "\' already defined")
```

```
17      else:
18          p.parser.var.update({p[1] : p.parser.startup})
19          p.parser.startup += 1
20      p[0] = p[3]
21
22
23  def p_Attr_value(p):
24      "Attr : id '=' Atomic ',' Attr"
25      if type(p.parser.var.get(p[1])):
26          print("Semantic error:")
27          print("\tVariable \'" + p[1] + "\' already defined")
28          p[0] = p[5]
29      else:
30          p.parser.var.update({p[1] : p.parser.startup})
31          p.parser.startup += 1
32          p[0] = 1 + p[5]
33          outputFile.write("PUSHI " + str(p[3]) + "\n")
34
35  def p_Attr_value_empty(p):
36      "Attr : id '=' integer ';'"
37      if type(p.parser.var.get(p[1])):
38          p.parser.var.update({p[1] : p.parser.startup})
39          p.parser.startup += 1
40          p[0] = 1
41          outputFile.write("PUSHI " + str(p[3]) + "\n")
42      else:
43          print("Semantic error:")
44          print("\tVariable \'" + p[1] + "\' already defined")
45          p[0] = 0
46
47  def p_Attr_empty(p):
48      "Attr : id ';'"
49      if type(p.parser.var.get(p[1])):
50          p.parser.var.update({p[1] : p.parser.startup})
51          p.parser.startup += 1
52      else:
53          print("Semantic error:")
54          print("\tVariable \'" + p[1] + "\' already defined")
55      p[0] = 0
56
57
58  def p_Line_Read_Array(p):
59      "Line : read '(' id '[' integer ']' ')'"
60      id = p.parser.var.get(p[3])
61      if type(id) is tuple:
```

```
62          outputFile.write("\tPUSHGP\n\tPUSHI " + str(id[0]) +
    "\n\tPADD\n")
63          outputFile.write("\tPUSHI " + str(p[5]) + "\n")
64          outputFile.write("\tREAD\n")
65          outputFile.write("\tATOI\n")
66          outputFile.write("\tSTOREN\n")
67      elif type(id):
68          print("Semantic error:")
69          print("\tVariable \'" + p[3] + "\' is not an array.")
70      else:
71          print("Semantic error:")
72          print("\tVariable \'" + p[3] + "\' not defined.")
73
74
75
76  def p_Line_Read_Array_Id(p):
77      "Line : read '(' id '[' id ']' ')'"
78      id = p.parser.var.get(p[3])
79      if type(id) is tuple:
80          id1 = p.parser.var.get(p[5])
81          if type(id1):
82              outputFile.write("\tPUSHGP\n\tPUSHI " + str(id
    [0]) + "\n\tPADD\n")
83              outputFile.write("\tPUSHG " + str(p.parser.var.
    get(p[5])) + "\n")
84              outputFile.write("\tREAD\n")
85              outputFile.write("\tATOI\n")
86              outputFile.write("\tSTOREN\n")
87          else:
88              print("Semantic error:")
89              print("\tVariable \'" + p[5] + "\' not defined.")
90      elif type(id):
91          print("Semantic error:")
92          print("\tVariable \'" + p[3] + "\' is not an array.")
93      else:
94          print("Semantic error:")
95          print("\tVariable \'" + p[3] + "\' not defined.")
96
97  def p_Line_Store_Attr(p):
98      "Line : id '=' Exp ';'"
99      id = p.parser.var.get(p[1])
100     if type(id) is int or type(id) is tuple:
101         outputFile.write("\tSTOREG " + str(p.parser.var.get(p
    [1])) + "\n")
102     else:
```

```
103         print("Semantic error:")
104         print("\tVariable \'" + p[1] + "\' not defined.")
105 
106 
107 # Production rules for Array
108 def p_Array(p):
109     "Array : id '[' integer ']'"
110     id = p.parser.var.get(p[1])
111     if type(id):
112         if type(id) is tuple:
113             outputFile.write("\tPUSHGP\n\tPUSHI " + str(id
    [0]) + "\n\tPADD\n")
114             outputFile.write("\tPUSHI " + str(p[3]) + "\n")
115         else:
116             outputFile.write("\tPUSHGP\n")
117             print("Semantic error:")
118             print("\tVariable \'" + p[1] + "\' is not an
    array.")
119     else:
120         outputFile.write("\tPUSHGP\n")
121         print("Semantic error:")
122         print("\tVariable \'" + p[1] + "\' not defined.")
123 
124 def p_Array_id(p):
125     "Array : id '[' id ']'"
126     id = p.parser.var.get(p[1])
127     id1 = p.parser.var.get(p[3])
128     if type(id):
129         if type(id) is tuple:
130             if type(id1) is int:
131                 outputFile.write("\tPUSHGP\n\tPUSHI " + str(
    id[0]) + "\n\tPADD\n")
132                 outputFile.write("\tPUSHG " + str(id1) + "\n"
    )
133             else:
134                 outputFile.write("\tPUSHGP\n")
135                 print("Semantic error:")
136                 print("\tVariable \'" + p[3] + "\' not
    defined.")
137         else:
138             outputFile.write("\tPUSHGP\n")
139             print("Semantic error:")
140             print("\tVariable \'" + p[1] + "\' is not an
    array.")
141     else:
```

```
142         outputFile.write("\tPUSHGP\n")
143         print("Semantic error:")
144         print("\tVariable \'" + p[1] + "\' not defined.")
145 
146 
147 def p_Atomic_Array(p):
148     "Atomic : id '[' integer ']'"
149     id = p.parser.var.get(p[1])
150     if type(id):
151         if type(id) is tuple:
152             outputFile.write("\tPUSHGP\n\tPUSHI " + str(id
    [0]) + "\n\tPADD\n")
153             outputFile.write("\tPUSHI " + str(p[3]) + "\n")
154             outputFile.write("\tLOADN\n")
155         else:
156             print("Semantic error:")
157             print("\tVariable \'" + p[1] + "\' is not an
    array.")
158     else:
159         print("Semantic error:")
160         print("\tVariable \'" + p[1] + "\' not defined.")
161 
162 def p_Atomic_Array_Id(p):
163     "Atomic : id '[' id ']'"
164     id = p.parser.var.get(p[1])
165     if type(id):
166         if type(id) is tuple:
167             if type(p.parser.var.get(p[3])) is int:
168                 outputFile.write("\tPUSHGP\n\tPUSHI " + str(
    id[0]) + "\n\tPADD\n")
169                 outputFile.write("\tPUSHG " + str(p.parser.
    var.get(p[3])) + "\n")
170                 outputFile.write("\tLOADN\n")
171             else:
172                 print("Semantic error:")
173                 print("\tVariable \'" + p[3] + "\' not
    defined.")
174         else:
175             print("Semantic error:")
176             print("\tVariable \'" + p[1] + "\' is not an
    array.")
177     else:
178         print("Semantic error:")
179         print("\tVariable \'" + p[1] + "\' not defined.")
```

3 Exemplos de utilização

3.1 MMC

```
int NumOne, NumTwo, maxValue;
start
    printS("Please Enter two integer Values \n")
    read(NumOne)
    read(NumTwo)

    if (NumOne > NumTwo) {
        maxValue = NumOne;
    } else{
        maxValue = NumTwo;
    }

    while(!maxValue / NumOne == NumTwo and maxValue / NumTwo
   == NumOne!)
    {
    maxValue++;
  }
    printS("LCM of " + NumOne + " and " + NumTwo + " = " +
   maxValue)
```

```
PUSHN 4
START
  PUSHS "Please Enter two integer Values \n"
  WRITES
  READ
  ATOI
  STOREG 2
  READ
  ATOI
  STOREG 1
  PUSHG 2
  PUSHG 1
  SUP
  JZ if1
  PUSHG 2
  STOREG 0
  JUMP else2
if1:
  PUSHG 1
  STOREG 0
else2:
```

```
while3:
  PUSHG 0
  PUSHG 2
  DIV
  PUSHG 1
  EQUAL
  PUSHG 0
  PUSHG 1
  DIV
  PUSHG 2
  EQUAL
  MUL
  NOT

  JZ whileEnd4
  PUSHG 0
  PUSHI 1
  ADD
  STOREG 0
  JUMP while3
whileEnd4:
  PUSHS "LCM of "
  WRITES
  PUSHG 2
  STRI
  WRITES
  PUSHS " and "
  WRITES
  PUSHG 1
  STRI
  WRITES
  PUSHS " = "
  WRITES
  PUSHG 0
  STRI
  WRITES
STOP
```

3.2 Triangulo

```
int size, curr;
start
    printS("Insira o tamanho do triangulo:\n")
    read(size)
```

```
    while(size > 0){
        curr = size;
        size --;
        while (curr > 0){
            printS("* ")
            curr --;
        }
        printS("\n")
    }
```

```
PUSHN 3
START
  PUSHS "Insira o tamanho do triangulo\n"
  WRITES
  READ
  ATOI
  STOREG 1
while1:
  PUSHG 1
  PUSHI 0
  SUP

  JZ whileEnd2
  PUSHG 1
  STOREG 0
  PUSHG 1
  PUSHI 1
SUB
  STOREG 1
while3:
  PUSHG 0
  PUSHI 0
  SUP

  JZ whileEnd4
  PUSHS "* "
  WRITES
  PUSHG 0
  PUSHI 1
SUB
  STOREG 0
  JUMP while3
whileEnd4:
  PUSHS "\n"
  WRITES
  JUMP while1
```

```
whileEnd2:
STOP
```

3.3 Lados de um quadrado

```
int a, b, c, d;
start
printS("Insira os lados do quadrado:\n")
read(a)
read(b)
read(c)
read(d)
if ( a == b and a == c and  a == d){
    printS("S  o  lados de um quadrado!!\n")
} else {
    printS("N  o  s  o  lados de um quadrado :(\n")
}
```

```
PUSHN 5
START
  PUSHS "Insira os lados do quadrado:\n"
  WRITES
  READ
  ATOI
  STOREG 3
  READ
  ATOI
  STOREG 2
  READ
  ATOI
  STOREG 1
  READ
  ATOI
  STOREG 0
  PUSHG 3
  PUSHG 2
  EQUAL
  PUSHG 3
  PUSHG 1
  EQUAL
  MUL
  PUSHG 3
  PUSHG 0
  EQUAL
```

```
  MUL
  JZ if1
  PUSHS "S o lados de um quadrado!!\n"
  WRITES
  JUMP else2
if1:
  PUSHS "N o s o lados de um quadrado :(\n"
  WRITES
else2:
STOP
```

3.4 Menor número de N números

```
int N;
int menor;
int este;
start
printS("Escreva o n mero de inteiros que pretende escrever.\
    n")
read(N)
printS("Insira os " + N +" n meros:\n")
N--;
read(menor)
repeat(N){
    read(este)
    if( este < menor ){
        menor = este;
    }
}
printS("Menor n mero: " + menor + "\n")
```

```
PUSHN 4
START
  PUSHS "Escreva o n mero de inteiros que pretende escrever
    .\n"
  WRITES
  READ
  ATOI
  STOREG 0
  PUSHS "Insira os "
  WRITES
  PUSHG 0
  STRI
  WRITES
```

```
  PUSHS " n meros:\n"
  WRITES
  PUSHG 0
  PUSHI 1
SUB
  STOREG 0
  READ
  ATOI
  STOREG 1
  PUSHG 0
  STOREG 3
repeat1:
  PUSHG 3
  PUSHI 0
  SUP
  JZ repeatEnd2
  READ
  ATOI
  STOREG 2
  PUSHG 2
  PUSHG 1
  INF
  JZ if3
  PUSHG 2
  STOREG 1
if3:
  PUSHG 3
  PUSHI 1
  SUB
  STOREG 3
  JUMP repeat1
repeatEnd2:
  PUSHS "Menor n mero: "
  WRITES
  PUSHG 1
  STRI
  WRITES
  PUSHS "\n"
  WRITES
STOP
```

4 Conclusão

5 Apêndice

5.1 Lex

```
# Lexer do compilador

import re
import ply.lex as lex

# \textit{token}s
reserved = {
    'print' : 'print',
    'printS' : 'printS',
    'start' : 'start',
    'read' : 'read',
    'if' : 'if',
    'else' : 'else',
    'and' : 'and',
    'or' : 'or',
    'while' : 'while',
    'repeat' : 'repeat',
    'int' : 'int',
}
\textit{token}s = ['integer','id', 'string'] + list(reserved.
    values())
literals = [
    '+','-','*','/','%',
    '(',')','[',']','{','}',
    ',',';','=','<','>','!']

def t_comment(t):
    r'(\#[^\n]*)|(\#[^\#]*)'
    pass

def t_id(t):
    r'[a-zA-Z]+'
    t.type = reserved.get(t.value,'id')
    return t

def t_integer(t):
    r'\d+'
    t.value = int(t.value)
```

```
38      return t
39
40  def t_string(t):
41      r'\"[^\"]*\"'
42      return t
43
44  def t_newline(t):
45      r'\n'
46      t.lexer.lineno += 1
47
48  t_ignore = " \t"
49
50  def t_error(t):
51      print('LEX ERROR:\n\t' + str(t))
52      print("\tIllegal character \'" + t.value[0] + '\'')
53      print("\tLine " + str(t.lexer.lineno))
54      t.lexer.skip(1)
55
56  lexer = lex.lex()
```

5.2 YACC

```
1   import ply.yacc as yacc
2   import re
3   # Get the \textit{token}s from the lexer
4   from compilador_lex import \textit{token}s
5   from difflib import SequenceMatcher
6
7   # Production rules for Program
8   def p_Program(p):
9       "Program : Start Code"
10      outputFile.write("STOP")
11
12  # Production rules for start
13  def p_Start(p):
14      "Start : Declaration start"
15      p.parser.var.update({"0" : p.parser.startup})
16      p.parser.startup += 1
17      outputFile.write("PUSHN " + str(p.parser.startup - p[1])
        + "\n")
18      outputFile.write("START\n")
19
20  # Production rules for declaration
21  def p_Declaration(p):
```

```
22      "Declaration : int Attr Declaration"
23      p[0] = p[2] + p[3]
24
25  def p_Declaration_Array(p):
26      "Declaration : int id '[' integer ']' ';' Declaration"
27      if id := p.parser.var.get(p[2]):
28          print("Semantic error:")
29          print("\tVariable \'" + p[2] + "\' already defined")
30      else:
31          p.parser.var.update({p[2] : (p.parser.startup, p[4])
    })
32          p.parser.startup += p[4]
33      p[0] = p[7]
34
35
36  def p_Declaration_Empty(p):
37      "Declaration : "
38      p[0] = 0
39
40  # Production rules for Attr
41  def p_Attr(p):
42      "Attr : id ',' Attr "
43      if type(p.parser.var.get(p[1])):
44          print("Semantic error:")
45          print("\tVariable \'" + p[1] + "\' already defined")
46      else:
47          p.parser.var.update({p[1] : p.parser.startup})
48          p.parser.startup += 1
49      p[0] = p[3]
50
51  def p_Attr_value(p):
52      "Attr : id '=' Atomic ',' Attr"
53      if type(p.parser.var.get(p[1])):
54          print("Semantic error:")
55          print("\tVariable \'" + p[1] + "\' already defined")
56          p[0] = p[5]
57      else:
58          p.parser.var.update({p[1] : p.parser.startup})
59          p.parser.startup += 1
60          p[0] = 1 + p[5]
61          outputFile.write("PUSHI " + str(p[3]) + "\n")
62
63  def p_Attr_value_empty(p):
64      "Attr : id '=' integer ';'"
65      if type(p.parser.var.get(p[1])):
```

```
66          p.parser.var.update({p[1] : p.parser.startup})
67          p.parser.startup += 1
68          p[0] = 1
69          outputFile.write("PUSHI " + str(p[3]) + "\n")
70      else:
71          print("Semantic error:")
72          print("\tVariable \'" + p[1] + "\' already defined")
73          p[0] = 0
74
75  def p_Attr_empty(p):
76      "Attr : id ';'"
77      if type(p.parser.var.get(p[1])):
78          p.parser.var.update({p[1] : p.parser.startup})
79          p.parser.startup += 1
80      else:
81          print("Semantic error:")
82          print("\tVariable \'" + p[1] + "\' already defined")
83      p[0] = 0
84
85  # Production rules for code
86  def p_Code(p):
87      "Code : Line Code"
88      pass
89
90  def p_Code_empty(p):
91      "Code : "
92      pass
93
94  # Production rules for line
95  def p_Line_Print_Exp(p):
96      "Line : print '(' Exp ')'"
97      outputFile.write("\tWRITEI\n")
98
99  def p_Line_Print_String(p):
100     "Line : printS '(' String ')'"
101
102 def p_Line_Read_Attr(p):
103     "Line : read '(' id ')'"
104     outputFile.write("\tREAD\n")
105     outputFile.write("\tATOI\n")
106     outputFile.write("\tSTOREG " + str(p.parser.var.get(p[3]))
    ) + "\n")
107
108 def p_Line_Read_Array(p):
109     "Line : read '(' id '[' integer ']' ')'"
```

```
110     id = p.parser.var.get(p[3])
111     if type(id) is tuple:
112         outputFile.write("\tPUSHGP\n\tPUSHI " + str(id[0]) +
    "\n\tPADD\n")
113         outputFile.write("\tPUSHI " + str(p[5]) + "\n")
114         outputFile.write("\tREAD\n")
115         outputFile.write("\tATOI\n")
116         outputFile.write("\tSTOREN\n")
117     elif type(id):
118         print("Semantic error:")
119         print("\tVariable \'" + p[3] + "\' is not an array.")
120     else:
121         print("Semantic error:")
122         print("\tVariable \'" + p[3] + "\' not defined.")
123 
124 
125 
126 def p_Line_Read_Array_Id(p):
127     "Line : read '(' id '[' id ']' ')'"
128     id = p.parser.var.get(p[3])
129     if type(id) is tuple:
130         id1 = p.parser.var.get(p[5])
131         if type(id1):
132             outputFile.write("\tPUSHGP\n\tPUSHI " + str(id
    [0]) + "\n\tPADD\n")
133             outputFile.write("\tPUSHG " + str(p.parser.var.
    get(p[5])) + "\n")
134             outputFile.write("\tREAD\n")
135             outputFile.write("\tATOI\n")
136             outputFile.write("\tSTOREN\n")
137         else:
138             print("Semantic error:")
139             print("\tVariable \'" + p[5] + "\' not defined.")
140     elif type(id):
141         print("Semantic error:")
142         print("\tVariable \'" + p[3] + "\' is not an array.")
143     else:
144         print("Semantic error:")
145         print("\tVariable \'" + p[3] + "\' not defined.")
146 
147 def p_Line_Store_Attr(p):
148     "Line : id '=' Exp ';'"
149     id = p.parser.var.get(p[1])
150     if type(id) is int or type(id) is tuple:
151         outputFile.write("\tSTOREG " + str(p.parser.var.get(p
```

```
    [1])) + "\n")
152     else:
153         print("Semantic error:")
154         print("\tVariable \'" + p[1] + "\' not defined.")
155 
156 def p_Line_Store_Attr_Array(p):
157     "Line : Array '=' Exp ';'"
158     outputFile.write("\tSTOREN\n")
159 
160 def p_Line_Inc_Attr(p):
161     "Line : id '+' '+' ';'"
162     outputFile.write("\tPUSHG " + str(p.parser.var.get(p[1],
    0)) + "\n")
163     outputFile.write("\tPUSHI 1\n\tADD\n")
164     outputFile.write("\tSTOREG " + str(p.parser.var.get(p[1])
    ) + "\n")
165 
166 def p_Line_Dec_Attr(p):
167     "Line : id '-' '-' ';'"
168     outputFile.write("\tPUSHG " + str(p.parser.var.get(p[1],
    0)) + "\n")
169     outputFile.write("\tPUSHI 1\nSUB\n")
170     outputFile.write("\tSTOREG " + str(p.parser.var.get(p[1])
    ) + "\n")
171 
172 def p_Line_Cond(p):
173     "Line : if Ifcond CondCode"
174     pass
175 
176 def p_Line_While(p):
177     "Line : WhileStart WhileLoop '{' Code '}'"
178     outputFile.write("\tJUMP " + p[1] + "\n")
179     outputFile.write(p[2] + ":\n")
180 
181 def p_Line_Repeat(p):
182     "Line : RepeatStart RepeatLoop '{' Code '}'"
183     outputFile.write("\tPUSHG " + str(p.parser.var.get("0"))
    + "\n\tPUSHI 1\n\tSUB\n")
184     outputFile.write("\tSTOREG " + str(p.parser.var.get("0"))
    + "\n")
185     outputFile.write("\tJUMP " + p[1] + "\n")
186     outputFile.write(p[2] + ":\n")
187 
188 def p_Line_Exp(p):
189     "Line : '(' Exp ')'"
```

```
190     pass
191 
192 # Production for WhileStart
193 def p_WhileStart(p):
194     "WhileStart : while"
195     p.parser.id += 1
196     p[0] = "while" + str(p.parser.id)
197     outputFile.write(p[0] + ":\n")
198 
199 # Production for WhileLoop
200 def p_WhileLoop(p):
201     "WhileLoop : '(' Cond ')'"
202     p.parser.id += 1
203     p[0] = "whileEnd" + str(p.parser.id)
204     outputFile.write("\tJZ " + p[0] + "\n")
205 
206 # Production for WhileLoop
207 def p_WhileLoop_error(p):
208     "WhileLoop : '(' error ')'"
209     print("\tError found in 'while' condition.")
210     print("\tExpected while condition:\n\t\twhile(condition){
    code}")
211     print("\tExpected condition structure:")
212     print('''\t\tCondition : Exp </>/=</>=/==/!= Exp
213                 Condition and Condition
214                 Condition or Condition
215                 !Condition!''')
216     outputFile.write("\tPUSHI 0\n")
217     p.parser.id += 1
218     p[0] = "whileEnd" + str(p.parser.id)
219     outputFile.write("\tJZ " + p[0] + "\n")
220 
221 # Production for Repeat
222 def p_RepeatStart(p):
223     "RepeatStart : repeat '(' Exp"
224     outputFile.write("\tSTOREG " + str(p.parser.var.get("0"))
    + "\n")
225     p.parser.id += 1
226     p[0] = "repeat" + str(p.parser.id)
227     outputFile.write(p[0] + ":\n")
228 
229 def p_RepeatStart_error(p):
230     "RepeatStart : repeat '(' error"
231     outputFile.write("\tPUSHI 0\n")
232     outputFile.write("\tSTOREG " + str(p.parser.var.get("0"))
```

```
    + "\n")
    p.parser.id += 1
    p[0] = "repeat" + str(p.parser.id)
    outputFile.write(p[0] + ":\n")
    print("\tError found in repeat number.")
    print("\tValid repeat structure:\n\t\trepeat(integer){
   code}")
    print("\tPlease insert valid integer.")

# Production for RepeatLoop
def p_RepeatLoop(p):
    "RepeatLoop : ')'"
    outputFile.write("\tPUSHG " + str(p.parser.var.get("0"))
   + "\n\tPUSHI 0\n\tSUP\n")
    p.parser.id += 1
    p[0] = "repeatEnd" + str(p.parser.id)
    outputFile.write("\tJZ " + p[0] + "\n")

# Production rules for Ifcond
def p_Ifcond(p):
    "Ifcond : '(' Cond ')'"
    pass

def p_Ifcond_error(p):
    "Ifcond : '(' error ')'"
    print("\tError found in 'if' condition.")
    print("\tExpected if structure:\n\t\tif(condition){code}\
   n\t\tif(condition){code}else{code}")
    print("\tExpected condition structure:")
    print('''\t\tCondition : Exp </>/=</>=/==/!= Exp
                Condition and Condition
                Condition or Condition
                !Condition!''')
    outputFile.write("\tPUSHI 0\n")

# Production for CondCode
def p_CondCode(p):
    "CondCode : IfStart Code '}'"
    outputFile.write(p[1] + ":\n")

def p_CondCode_Else(p):
    "CondCode : ElseStart Code '}'"
    outputFile.write(p[1] + ":\n")

# Production rules for Ifstart
```

```
274 def p_IfStart(p):
275     "IfStart : '{'"
276     p.parser.id += 1
277     p[0] = "if" + str(p.parser.id)
278     outputFile.write("\tJZ " + p[0] + "\n")
279 
280 def p_ElseStart(p):
281     "ElseStart : IfStart Code '}' else '{'"
282     p.parser.id += 1
283     p[0] = "else" + str(p.parser.id)
284     outputFile.write("\tJUMP " + p[0] + "\n")
285     outputFile.write(p[1] + ":\n")
286 
287 # Production rules for Cond
288 # OR
289 def p_Cond_Or(p):
290     "Cond : Cond or CondAnd"
291     outputFile.write("\tADD\n")
292 
293 def p_Cond_Empty(p):
294     "Cond : CondAnd"
295     pass
296 
297 # AND
298 def p_CondAnd(p):
299     "CondAnd : CondAnd and CondNot"
300     outputFile.write("\tMUL\n")
301 
302 def p_CondAnd_empty(p):
303     "CondAnd : CondNot"
304     pass
305 
306 # Not Condition
307 def p_CondNot(p):
308     "CondNot : '!' Cond '!'"
309     outputFile.write("\tNOT\n")
310 
311 def p_CondNot_Par(p):
312     "CondNot : '(' Cond ')'"
313     pass
314 
315 def p_CondNot_Empty(p):
316     "CondNot : Rel"
317     pass
318 
```

```
319 # Production rules for Rel
320 def p_Rel_Sup(p):
321     "Rel : Exp '>' Exp"
322     outputFile.write("\tSUP\n")
323 
324 def p_Rel_Supeq(p):
325     "Rel : Exp '>' '=' Exp"
326     outputFile.write("\tSUPEQ\n")
327 
328 def p_Rel_Inf(p):
329     "Rel : Exp '<' Exp"
330     outputFile.write("\tINF\n")
331 
332 def p_Rel_Infeq(p):
333     "Rel : Exp '=' '<' Exp"
334     outputFile.write("\tINFEQ\n")
335 
336 def p_Rel_Equal(p):
337     "Rel : Exp '=' '=' Exp"
338     outputFile.write("\tEQUAL\n")
339 
340 def p_Rel_Not_Equal(p):
341     "Rel : Exp '!' '=' Exp"
342     outputFile.write("\tEQUAL\n")
343     outputFile.write("\tNOT\n")
344 
345 # Production rules for Array
346 def p_Array(p):
347     "Array : id '[' integer ']'"
348     id = p.parser.var.get(p[1])
349     if type(id):
350         if type(id) is tuple:
351             outputFile.write("\tPUSHGP\n\tPUSHI " + str(id
    [0]) + "\n\tPADD\n")
352             outputFile.write("\tPUSHI " + str(p[3]) + "\n")
353         else:
354             outputFile.write("\tPUSHGP\n")
355             print("Semantic error:")
356             print("\tVariable \'" + p[1] + "\' is not an
    array.")
357     else:
358         outputFile.write("\tPUSHGP\n")
359         print("Semantic error:")
360         print("\tVariable \'" + p[1] + "\' not defined.")
361 
```

```
362 def p_Array_id(p):
363     "Array : id '[' id ']'"
364     id = p.parser.var.get(p[1])
365     id1 = p.parser.var.get(p[3])
366     if type(id):
367         if type(id) is tuple:
368             if type(id1) is int:
369                 outputFile.write("\tPUSHGP\n\tPUSHI " + str(
    id[0]) + "\n\tPADD\n")
370                 outputFile.write("\tPUSHG " + str(id1) + "\n"
    )
371             else:
372                 outputFile.write("\tPUSHGP\n")
373                 print("Semantic error:")
374                 print("\tVariable \'" + p[3] + "\' not
    defined.")
375         else:
376             outputFile.write("\tPUSHGP\n")
377             print("Semantic error:")
378             print("\tVariable \'" + p[1] + "\' is not an
    array.")
379     else:
380         outputFile.write("\tPUSHGP\n")
381         print("Semantic error:")
382         print("\tVariable \'" + p[1] + "\' not defined.")
383
384 # INTEGERS
385
386 # Production rules for exp
387 def p_Exp_SUM(p):
388     "Exp : Exp '+' Termo"
389     outputFile.write("\tADD\n")
390
391 def p_Exp_SUB(p):
392     "Exp : Exp '-' Termo"
393     outputFile.write("\tSUB\n")
394
395 def p_Exp(p):
396     "Exp : Termo"
397     pass
398
399 # Production rules for termo
400 def p_Termo_MULT(p):
401     "Termo : Termo '*' Factor"
402     outputFile.write("\tMUL\n")
```

```
403
404 def p_Termo_DIV(p):
405     "Termo : Termo '/' Factor"
406     outputFile.write("\tDIV\n")
407
408 def p_Termo_MOD(p):
409     "Termo : Termo '%' Factor"
410     outputFile.write("\tMOD\n")
411
412 def p_Termo(p):
413     "Termo : Factor"
414     pass
415
416 # Production rules for fator
417 def p_Factor(p):
418     "Factor : '(' Exp ')'"
419     pass
420
421 def p_Factor_Int(p):
422     "Factor : Atomic"
423     pass
424
425 def p_Factor_Signal(p):
426     "Factor : Signal Atomic"
427     outputFile.write("\tMUL\n")
428
429 # Production rules for Atomic
430 def p_Atomic_Int(p):
431     "Atomic : integer"
432     outputFile.write("\tPUSHI " + str(p[1])+ "\n")
433
434 def p_Atomic_Id(p):
435     "Atomic : id"
436     id = p.parser.var.get(p[1])
437     if type(id):
438         outputFile.write("\tPUSHG " + str(id) + "\n")
439     else:
440         print("Semantic error:")
441         print("\tVariable \'" + p[1] + "\' not defined.")
442
443 def p_Atomic_Array(p):
444     "Atomic : id '[' integer ']'"
445     id = p.parser.var.get(p[1])
446     if type(id):
447         if type(id) is tuple:
```

```
448                 outputFile.write("\tPUSHGP\n\tPUSHI " + str(id
    [0]) + "\n\tPADD\n")
449                 outputFile.write("\tPUSHI " + str(p[3]) + "\n")
450                 outputFile.write("\tLOADN\n")
451             else:
452                 print("Semantic error:")
453                 print("\tVariable \'" + p[1] + "\' is not an
    array.")
454         else:
455             print("Semantic error:")
456             print("\tVariable \'" + p[1] + "\' not defined.")
457 
458 def p_Atomic_Array_Id(p):
459     "Atomic : id '[' id ']'"
460     id = p.parser.var.get(p[1])
461     if type(id):
462         if type(id) is tuple:
463             if type(p.parser.var.get(p[3])) is int:
464                 outputFile.write("\tPUSHGP\n\tPUSHI " + str(
    id[0]) + "\n\tPADD\n")
465                 outputFile.write("\tPUSHG " + str(p.parser.
    var.get(p[3])) + "\n")
466                 outputFile.write("\tLOADN\n")
467             else:
468                 print("Semantic error:")
469                 print("\tVariable \'" + p[3] + "\' not
    defined.")
470         else:
471             print("Semantic error:")
472             print("\tVariable \'" + p[1] + "\' is not an
    array.")
473     else:
474         print("Semantic error:")
475         print("\tVariable \'" + p[1] + "\' not defined.")
476 
477 # Production rules for signal
478 def p_Signal_End(p):
479     "Signal : '-'"
480     outputFile.write("\tPUSHI -1\n")
481 
482 def p_Signal(p):
483     "Signal : Signal '-'"
484     outputFile.write("\tPUSHI -1\n")
485     outputFile.write("\tMUL\n")
486 
```

```
487 # Production rules for String
488 def p_String_str(p):
489     "String : String '+' string"
490     outputFile.write("\tPUSHS " + p[3] + "\n")
491     outputFile.write("\tWRITES\n")
492 
493 def p_String_int(p):
494     "String : String '+' id "
495     outputFile.write("\tPUSHG " + str(p.parser.var.get(p[3]))
    + "\n\tSTRI\n\tWRITES\n")
496 
497 def p_String_str_end(p):
498     "String : string"
499     outputFile.write("\tPUSHS " + p[1] + "\n")
500     outputFile.write("\tWRITES\n")
501 
502 def p_String_int_end(p):
503     "String : id "
504     outputFile.write("\tPUSHG " + str(p.parser.var.get(p[1]))
    + "\n\tSTRI\n")
505     outputFile.write("\tWRITES\n")
506 
507 # Error rule for syntax error
508 def p_error(p):
509     print("YACC ERROR:\n\t" + str(p))
510     line_size = p.lexpos - code.rfind('\n', 0, p.lexpos) +1
511     print("\tError found on line", p.lineno, "column",
    line_size)
512     if p.type == "id":
513         if expected := mostSimilar(p.value):
514             print("\tYou wrote \'" + p.value + "\', did you
    mean to write \'" + expected + "\'?")
515         else:
516             print("\tUnkown simbol: " + p.value)
517             pass
518     elif p.type == "start":
519         print("\tKeyword 'start' misused, this keyword is
    utilized in the begining of your code block.")
520     elif p.type == "print":
521         print("\tKeyword 'print' misused, this keyword prints
     any integer in the standard output.")
522     elif p.type == "printS":
523         print("\tKeyword 'printS' misused, this keyword
    prints any string of characters in the standard output.")
524     elif p.type == "print":
```

```
525         print("\tKeyword 'read' misused, this keyword reads
    any integer in the standard input.")
526     elif p.type == "if":
527         print("\tKeyword 'if' misused, this keyword initiates
     an if statement.")
528     elif p.type == "else":
529         print("\tKeyword 'else' misused, this keyword must be
     utilized after an if code block.")
530     elif p.type == "repeat":
531         print("\tKeyword 'repeat' misused, this keyword loops
     the given block of code as many times as the result of
    the given expression.")
532     elif p.type == "while":
533         print("\tKeyword 'while' misused, this keyword loops
    the given block of code as long as the given condition is
    true.")
534     elif p.type == "and":
535         print("\tKeyword 'and' misused, this keyword is the
    logical conjunction opetator.")
536     elif p.type == "or":
537         print("\tKeyword 'or' misused, this keyword is the
    logical disjunction opetator.")
538     elif p.type == "int":
539         print("\tKeyword 'int' misused, this keyword
    indicates the type of a variable.")
540     else:
541         pass
542
543 def mostSimilar(error):
544     r = (None,0.4)
545     for t in \textit{token}s:
546         if t != "id" and t != "integer" and t != "string":
547             sim = similar(error, t)
548             if r[1] < sim:
549                 r = (t,sim)
550     return r[0]
551
552 def similar(error,\textit{token}):
553     return SequenceMatcher(None, error, \textit{token}).ratio
    ()
554
555 # Build parser
556 parser = yacc.yacc()
557
558 import sys
```

```
559 if len(sys.argv) == 2 and re.search(r'\.txt$',sys.argv[1]):
560     path = sys.argv[1]
561 else:
562     path = "test.txt"
563 
564 inputFile = open(path,"r")
565 path = re.sub(r'\.txt$',
566         r'.vm',
567         path)
568 outputFile = open(path,"w")
569 
570 # parser state
571 parser.var = {}
572 parser.id = 0
573 parser.startup = len(parser.var)
574 
575 # Read input and parse the whole file
576 code = inputFile.read()
577 parser.parse(code)
578 
579 print("Exited parser")
580 
581 inputFile.close
582 outputFile.close
```