Processamento de Linguagens (3º ano MIEI)

Trabalho Prático 2

Relatório de Desenvolvimento

Alexandre Gomes (a89549)
Henrique Ribeiro (a89582)
José Duarte Alves (a89563)

30 de maio de 2021

Conteúdo

Capítulo 1

Introdução

1.1 Problema e Objetivos

Neste relatório será apresentado o trabalho realizado pelo grupo 41 no âmbito da disciplina de Processamento de Linguagens. A realização deste projeto visa dar uma opurtunidade de aprofundar o conhecimento sobre os vários conceitos leccionados na disciplina de PL, nomeadamente a engenharia de linguagens, a programação gramatical, a escrita de gramáticas, geradores de analisadores léxicos e geradores de compiladores baseados em gramáticas tradutoras. Para este efeito foi então proposto um exercício que consiste no desenvolvimento de um compilador que gera código para uma máquina virtual.

1.2 Estrutura do Relatório

Primeiramente começamos por descrever o problema e listar os requesitos a alcançar, de seguida no capítulo 3 mostramos a nossa solução para o analizador léxico, gramática independente de contexto e por fim para a gramática tradutora. Depois temos um conjunto de testes e seus respetivo resultado em código *assembly* da máquina virtual. Por fim temos um sucinta conclusão para o trabalho e o código total escrito da nossa solução.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

Desenvolver um compilador para uma linguagem criada por nós que vamos passar a chamar neoC, usando como recurso uma gramática tradutora e uma máquina virtual.

2.2 Especificação do Requisitos

- Definir uma linguagem de programação imperativa simples
  - Declarar variáveis locais inteiras
  - Declarar variáveis globais inteiras
  - Declarar e manusear variáveis estruturados do tipo array (1 ou 2 dimensões) de inteiros.
  - Efetuar instruções algorítmicas básicas
  - Efetuar instruções algorítmicas especiais
  - Efetuar instruções condicionais
  - Ler e escrever do standard input
  - efetuar instruções cíclicas para controlo de fluxo de execução, per mitindo o seu aninhamento (while-do, repeat(n) for-do)
  - Definir e invocar subprogramas que podem retornar um resultado do tipo inteiro.
- Analisador Léxico
- Gramática independente de contexto
- Gramática tradutora para código assembly

Capítulo 3

Concepção/desenho da Resolução

A construção do reconhecedor sintático da linguagem neoC, foi iniciada com a construção de um analisador léxico, de seguida uma gramática independente de contexto, que contêm a lógica associada à linguagem que se pretende reconhecer, e por fim, através de uma gramática tradutora, é gerada o código assembly com a solução.

3.1 Analisador Léxico

O analisador léxico tem como função traduzir a sequência de caracteres de entrada num conjunto de símbolos léxicos(tokens) que formam a linguagem a reconhecer. Com apoio ao módulo PLY.FLEX da linguagem de programação Python alguns destes símbolos foram iniciados como literais, estes são tokens com apenas um character e cuja sua tradução é ele mesmo.

```
tokens = ['num', 'id', 'int','print','println','prints','string','main','repeat','read','sup',
'inf','eq','supeq','infeq','not','diff','and','or','if','else','for','while','plus','addeq',
'subeq','diveq','muleq','addeql','subeql','diveql','muleql','minus','plusl','minusl','modeq',
'modeql','return','global','gid']

literals = ['=','(',')','{','}','+','*','-','/','[',']',';','%',',']

def t_numR(t):
    r'\d+\.\d+'
    t.value = float(t.value)
    return t

def t_num(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_string(t):
    r'"[^"]*"'
    return t

def t_diveql(t):
```

```
    r'\./='
    return t

def t_addeql(t):
    r'\.\+\='
    return t

def t_subeql(t):
    r'\.\-\='
    return t


def t_muleql(t):
    r'\.\*\='
    return t

def t_modeql(t):
    r'\.\%\='
    return t

def t_plus(t):
    r'\+\+'
    return t

def t_plusl(t):
    r'\.\+'
    return t

def t_minus(t):
    r'--'
    return t

def t_minusl(t):
    r'\.-'
    return t

def t_addeq(t):
    r'\+\='
    return t

def t_subeq(t):
    r'\-\='
    return t

def t_diveq(t):
    r'/='
    return t
```

```
def t_muleq(t):
    r'\*\='
    return t

def t_modeq(t):
    r'\%\='
    return t

def t_supeq(t):
    r'>='
    return t

def t_infeq(t):
    r'<='
    return t

def t_sup(t):
    r'>'
    return t

def t_inf(t):
    r'<'
    return t

def t_diff(t):
    r'!='
    return t

def t_not(t):
    r'!'
    return t

def t_eq(t):
    r'=='
    return t

def t_and(t):
    r'&&'
    return t

def t_or(t):
    r'\|\|'
    return t

def t_gid(t):
    r'&[a-zA-Z]\w*'
    return t
```

```
def t_id(t):
    r'[a-zA-Z]\w*'
    return t
```

Os símbolos restantes tem todos como sua tradução o seu próprio nome por exemplo:

```
def t_main(t):
    r'main'
    return t
```

3.2 Gramática Independente de Contexto

O próximo passo para a implementação do compilador, foi criar uma linguagem independente do contexto, esta vai ser reconhecida através do módulo PLY.YACC, sendo assim to tipo *bottom-up*.
Um programa pode ser dividido em três blocos, no primeiro bloco encontram-se as declarações das variáveis globais, o segundo bloco encontra-se a declaração a função main principal e no último bloco encontram-se as defenições de funções auxiliares.

```
Prog -> GlobalBlc MainBlc DefsBlcs
```

O bloco de variáveis globais começa com uma inicialização com o símbolo léxico global e depois contêm as diversas declarações de variáveis.

```
GlobalBlc -> GlobalBegin VarBlcs
           | &

GlobalBegin -> global '{'
```

A função principal começa com a inicialização com o símbolo léxico main e depois contêm um conjunto de declarações de variáveis e um conjunto de instruções.

```
MainBlc -> main '{' VarBlcs Insts '}'
```

O conjunto de declarações de funções auxiliares é composto por um conjunto de funções cuja a sua inicialização é determinada pelo utilizador com auxilio a um id.

```
DefBlcs -> DefBlcs DefBlc
         | &

DefBlc | id '{' VarBlcs Insts '}'
```

Um bloco de variáveis pode ter um número indeterminado de blocos de interios, estes por sua vês podem ter um número indeterminado de declarações e tem de ser iniciados com o símbolo léxico int.

```
VarBlcs -> VarBlcs BlcInt
         | &

BlcInt -> int '{' Dcls '}'

Dcls -> Dcls Dcl
      | &
```

Variáveis podem ser declaradas de várias formas, pode ser declarado um inteiro ou array de inteiros(uni ou bi dimensional) sem lhe atribuir qualquer valor, também podem ser atribuídos valores sendo que para um array é possível atribuir a todos os elementos o mesmo valor ou um conjunto de valores não sendo necessário indicar o seu tamanho.

```
Dcl -> id
     | id '=' num
     | id '[' num ']'
     | id '[' num ']' '=' num
     | id '[' ']' '=' '{' Nums '}'
     | id '[' num ']' '[' num ']'
     | id '[' num ']' '[' num ']' '=' num
     | id '[' ']' '[' ']' '=' '{' BlcsNums '}'

BlcsNums -> BlcsNums '{' Nums '}'
          | &

Nums -> Nums num
      | &
```

Depois das declarações de variáveis um programa é essencialmente composto por um conjunto de instruções.

```
Insts -> Insts Inst
Insts  | &

Inst -> Exp
      | Attr
      | Print
      | Println
      | Prints
      | Read
      | Repeat
      | For
      | While
      | If
      | Return
```

As expressões são uma componente fulcral da linguagem de programação, uma expressão pode conter um termo que por sua vez pode conter um factor isto é necessáio visto que existem operações com diferentes prioridades, para além das operações normais como a soma subtração divisão e multiplicação existem também

operações especiais. Apesar de não ser tanto comum também demos a possibilidade ao utilizador de poder usar condições como fatores para as expressões visto que estas tem sempre um resultado final (1 ou 0).

```
Exp -> Exp '+' Term
     | Exp '-' Term
     | id addeq Term          #Soma do termo a id guardando o resultado em id
     | id subeq Term          #Subtração do termo a id guardando o resultado em id
     | id addeql Term         #Igual ao addeq mas deixa o valor antigo de id na stack
     | id subeql Term         #Igual ao subeql mas deixa o valor antigo de id na stack
     | gid addeq Term
     | gid subeq Term
     | gid addeql Term
     | gid subeql Term
     | Term

Term -> Term '*' Factor
      | Term '/' Factor
      | Term '\%' Factor
      | id muleq Factor       #Multiplicação do termo com id guardando o resultado em id
      | id diveq Factor       #Divisão do termo com id guardando o resultado em id
      | id modeq Factor       #Resto da divisão do termo com id guardando o resultado em id
      | id muleql Factor      #Igual a muleq mas deixa o valor antigo de id na stack
      | id diveql Factor      #Igual a diveq mas deixa o valor antigo de id na stack
      | id modeql Factor      #Igual a modeq mas deixa o valor antigo de id na stack
      | gid muleq Factor
      | gid diveq Factor
      | gid modeq Factor
      | gid muleql Factor
      | gid diveql Factor
      | gid modeql Factor
      | Factor

Factor -> id
        | id '[' Exp ']'
        | id '[' Exp ']' '[' Exp ']'
        | gid
        | gid '[' Exp ']'
        | gid '[' Exp ']' '[' Exp ']'
        | num
        | id plus             #Soma 1 ao id guardando o resultado em id
        | id minus            #Subtrai 1 ao id guardando o resultado em id
        | id plusl            #Igual a plus mas deixa o valor antigo de id na stack
        | id minusl           #Igual a minus mas deixa o valor antigo de id na stack
        | gid plus
        | gid plusl
        | gid minus
        | gid minusl
        | id '(' ')'          #Chamada de uma função
```

```
        | '(' Cond ')'
        | '(' Exp ')'
```

Existem seis tipos diferentes de atribuições, primeiro é sempre necessário o id da variável, visto que estas podem ser locais ou globais então existem dois tipos de símbolos léxicos para representar este id, id para locais e gid para globais, cada uma destas variáveis pode também ser do tipo inteiro ou array (uni ou bi dimensional) de inteiros. Pode se fazer a atribuição de qualquer expressão a qualquer variável, também pode se usar uma expressão para a representação do id nas variáveis do tipo array.

```
Attr -> id '=' Exp
      | id '[' Exp ']' '=' Exp
      | id '[' Exp ']' '[' Exp ']' '=' Exp
      | gid '=' Exp
      | gid '[' Exp ']' '=' Exp
      | gid '[' Exp ']' '[' Exp ']' '=' Exp
```

Para além das expressões e atribuições uma instrução também pode ser a chamada de funções predefinidas, que são responsáveis pela leitura do standard input, escrita no standar output de expressões (podendo ter um new line no fim) ou strings e o retorno de uma expressão que é util para as funções auxiliares.

```
Read -> read '(' id ')'

Print -> print '(' Exp ')'

Println -> println '(' Exp ')'

Prints -> prints '(' string ')'

Return -> return '(' Exp ')'
```

De seguida temos as instruções condicionais, a instrução if tem de ser iniciada com o símbolo léxico if, seguida de uma condição e um conjunto de instruções, pode ter ou não um else statement sendo que este tem ser iniciado por o seu símbolo léxico else e também é seguido de um conjunto de instruções.

```
If -> if '(' Cond ')' '{' Insts '}'
    | if '(' Cond ')' '{' Insts '}' else '{' Insts '}'"

Cond -> '(' Cond and Cond ')'
      | '(' Cond or Cond ')'
      | Exp sup Exp
      | Exp inf Exp
      | Exp supeq Exp
      | Exp infeq Exp
      | not Exp
      | Exp eq Exp
      | Exp diff Exp
```

As únicas instruções que faltam enunciar são as instruções cíclicas. O ciclo for tem de ser iniciado com o símbolo léxico que o identifica for, de seguida pode ter um conjunto de instruções seguidas de uma condição e mais dois conjuntos de instruções. O ciclo while tem de ser iniciado com o símbolo léxico que o identifica while, de seguida tem de ter uma condição e um conjunto de instruções. Finalmente o ciclo repeat que também tem de ser iniciado com o seu símbolo léxico repeat, deve ser seguido de uma expressão que indica quantas vezes as instruções tem de ser repetidas e de um conjunto de instruções.

```
For -> for '(' Insts ';' Cond ';' Insts ')' '{' Insts '}'

While -> while '(' Cond ')' '{' Insts '}'

Repeat -> repeat '(' Exp ')' '{' Insts '}'
```

3.3 Controlo de Estado

Devido á natureza do problema foi necessário fazer um controlo do estado do programa no compilador.

- Foi necessário ter duas tabelas de identificadores (parser.table e parser.tableG) que guardam as variáveis locais e globais declaradas pelo utilizador e o seu parser.offset na stack.
- Também foi necessário a criação de uma variável parser.level que guarda o nível de aninhamento de ciclos repeat, isto é necessário visto que tem de ser o compilador a fazer o controlo deste ciclo.
- A variável parser.currTag serve para indicar qual é a o número da label atual, visto que não podem existir tags com o mesmo nome.
- A variável parser.isGlobal indica se estão a ser declaradas variáveis globais.
- Por fim as variáveis parser.arraySize e parser.linhas que servem para calcular o tamanho de um array quando o utilizador o define apenas com os valores sem dar o seu tamanho.

3.4 Gramática Tradutora

O próximo passo foi através do modulo PLY.YACC da linguagem pyhton criar a gramática tradutora que tem como objetivo a tradução do código escrito pelo utilizador em neoC em código assembly para a máquina de stack virtual.
A estratégia utilizada para gerar o código assembly foi sempre que um símbolo é reduzido o seu resultado é escrito em p[0] de forma a que quando todo o programa for compilado apenas basta escrever o resultado que se é a soma dos útimos 3 simbolos da gramática.

3.4.1 Declarações de Variáveis

Começando pelas declarações, é necessário saber sempre se a variável a declarar é global ou não para saber que tabela usar para a guardar.
Sempre que é guardada uma variável o offset atual é guardado na tabela de identificadores e incrementado por um, desta forma para cada função saberemos qual a posição da variável em relação ao frame pointer, no caso de ser um array também é guardado o número de colunas deste (0 se for uni dimensional)
Uma variável pode ser iniciada sem valor neste caso ela é iniciada a zero o que corresponde a `pushi 0` em assembly, se por outro lado for iniciado com um número(num) então é iniciado com esse número `pushi num`

```
def p_Dcl_0(p):
    "Dcl : id"
    p[0]=("pushi 0\n")
    if p.parser.isGlobal:
        p.parser.tableG[p[1]]=p.parser.offset
    else:
        p.parser.table[p[1]]=p.parser.offset
    p.parser.offset+=1

def p_Dcl_num(p):
    "Dcl : id '=' num"
    p[0]="pushi "+str(p[3])+"\n"
    if p.parser.isGlobal:
        p.parser.tableG[p[1]]=p.parser.offset
    else:
        p.parser.table[p[1]]=p.parser.offset
    p.parser.offset+=1
```

Um array pode ser inicializado de várias maneiras.

- Se for declarado apenas o tamanho(n) do array é usada a instrução `pushn` para colocar n zeros na stack
- Se for atribuído um valor(num) a um array é usada a instrução `pushi num` n vezes.
- Se for atribuído um conjunto de valores tiramos proveito do símbolo terminal Nums para escrever a instrução `pushi num` sempre que é lido um número;

```
def p_Dcl_Arr(p):
    "Dcl : id '[' num ']'"
    p[0]="pushn "+str(p[3])+"\n"
    if p.parser.isGlobal:
        p.parser.tableG[p[1]]=(p.parser.offset,0)
    else:
        p.parser.table[p[1]]=(p.parser.offset,0)
    p.parser.offset+=p[3]

def p_Dcl_Arr_valEq(p):
    "Dcl : id '[' num ']' '=' num"
    p[0]=""
    for i in range(p[3]):
        p[0]+="pushi "+str(p[6])+"\n"

    if p.parser.isGlobal:
        p.parser.tableG[p[1]]=(p.parser.offset,0)
    else:
        p.parser.table[p[1]]=(p.parser.offset,0)
```

```
    p.parser.offset+=p[3]

def p_Dcl_Arr_val(p):
    "Dcl : id '[' ']' '=' '{' Nums '}'"
    p[0]=p[6]
    if p.parser.isGlobal:
        p.parser.tableG[p[1]]=(p.parser.offset,0)
    else:
        p.parser.table[p[1]]=(p.parser.offset,0)
    p.parser.offset+=p.parser.arraySize
    p.parser.arraySize=0


def p_Nums(p):
    "Nums : Nums num"
    p.parser.arraySize+=1
    p[0]=p[1]+"pushi "+str(p[2])+"\n"

def p_Nums_End(p):
    "Nums : "
    p[0]= ""
```

Os arrays bidimensionais também podem ser inicializados das mesmas maneira.

- Se for declarado apenas o tamanho(n e m) do array é usada a instrução `pushn` para colocar n*m zeros na stack
- Se for atribuído um valor(num) a um array é usada a instrução `pushi num` n*m vezes.
- Se for atribuído um conjunto de valores tiramos proveito do símbolo terminal BlcsNums para escrever a instrução `pushi num` sempre que é lido um número, temos de ter cuidado e guardar contar o número de número que o utilizador insere para poder deduzir o tamanho do array final para depois introduzir este valor na tabela de identificadores.

```
def p_Dcl_Arr_2D(p):
    "Dcl : id '[' num ']' '[' num ']'"
    p[0]="pushn "+str(p[3]*p[6])+"\n"
    if p.parser.isGlobal:
        p.parser.tableG[p[1][1:]]=(p.parser.offset,p[6])
    else:
        p.parser.table[p[1]]=(p.parser.offset,p[6])
    p.parser.offset+=p[3]*p[6]

def p_Dcl_Arr_2D_valEq(p):
    "Dcl : id '[' num ']' '[' num ']' '=' num"
    p[0]=""
```

```
    for i in range(p[3]*p[6]):
        p[0]+="pushi "+str(p[9])+"\n"

    if p.parser.isGlobal:
        p.parser.tableG[p[1][1:]]=(p.parser.offset,p[6])
    else:
        p.parser.table[p[1]]=(p.parser.offset,p[6])
    p.parser.offset+=p[3]*p[6]


def p_Dcl_Arr_2D_val(p):
    "Dcl : id '[' ']' '[' ']' '=' '{' BlcsNums '}' "
    p[0]=p[8]
    if p.parser.isGlobal:
        p.parser.tableG[p[1][1:]]=(p.parser.offset,int(p.parser.arraySize/p.parser.linhas))
    else:
        p.parser.table[p[1]]=(p.parser.offset,int(p.parser.arraySize/p.parser.linhas))
    p.parser.offset+=p.parser.arraySize
    p.parser.arraySize=0
    p.parser.linhas=0

def p_BlcsNums(p):
    "BlcsNums : BlcsNums '{' Nums '}'"
    p.parser.linhas+=1
    p[0]=p[1]+p[3]

def p_BlcsNums_End(p):
    "BlcsNums : "
    p[0]=""
```

3.4.2 Expressões

De seguida vamos mostrar como foi feita a tradução das expressões começando pelos fatores.
Na sua forma mais reduzida uma fator pode ser um id, número ou uma chamada de uma função.
Usando a instrução `pushl` conseguimos obter uma variável que esteja na stack apenas temos de saber o seu offset neste caso ele está na tabela de identificadores.
Se for um número apenas temos fazer `pushi` do número.

```
def p_Factor_id(p):
    "Factor : id"
    p[0] = "pushl "+str(p.parser.table[p[1]])+"\n"

def p_Factor_gid(p):
    "Factor : gid"
    p[0] = "pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
```

```
def p_Factor_num(p):
    "Factor : num"
    p[0] = "pushi "+str(p[1])+"\n"
```

Para aceder a um valor de um array declarado previamente tiramos proveito das instruções `pushfp` e `padd` que juntas com o valor do offset permitem aceder a esse valor na stack. No fim é usada a instrução `loadn` para colocar o valor na stack.
Para aceder a um valor de um array bidimensional é necessário calcular o offset da variável na stack apartir do número de colunas(n) e da linha(l) e coluna(c) indicada pelo utilizador, a fórmula para calcular este offset é l*n+c.

```
def p_Factor_arr(p):
    "Factor : id '[' Exp ']'"
    p[0]="pushfp\n"
    p[0]+="pushi "+str(p.parser.table[p[1]][0])+"\n"
    p[0]+="padd\n"+p[3]
    p[0]+="loadn\n"

def p_Factor_arr_2d(p):
    "Factor : id '[' Exp ']' '[' Exp ']'"
    p[0]="pushfp\n"
    p[0]+="pushi "+str(p.parser.table[p[1]][0])+"\n"
    p[0]+="padd\n"+str(p[3])
    p[0]+="pushi "+str(p.parser.table[p[1]][1])+"\n"
    p[0]+="mul\n"+str(p[6])
    p[0]+="add\n"+"loadn\n"

def p_Factor_arrg(p):
    "Factor : gid '[' Exp ']'"
    p[0]="pushgp\n"
    p[0]+="pushi "+str(p.parser.tableG[p[1][1:]][0])+"\n"
    p[0]+="padd\n"+p[3]
    p[0]+="loadn\n"

def p_Factor_arrg_2d(p):
    "Factor : gid '[' Exp ']' '[' Exp ']'"
    p[0]="pushgp\n"
    p[0]+="pushi "+str(p.parser.tableG[p[1][1:]][0])+"\n"
    p[0]+="padd\n"+str(p[3])
    p[0]+="pushi "+str(p.parser.tableG[p[1][1:]][1])+"\n"
    p[0]+="mul\n"+str(p[6])
    p[0]+="add\n"+"loadn\n"
```

De seguida temos um conjunto de instruções especiais para os fatores, que guardam o valor final na variável id e podem deixar na stack o novo valor deste id ou o valor anterior.

```
#Soma um
def p_Factor_id_plus(p):
    "Factor : id plus"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="add\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"

def p_Factor_id_plusl(p):
    "Factor : id plusl"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="add\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"

#Subtrai um
def p_Factor_id_minus(p):
    "Factor : id minus"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="sub\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"

def p_Factor_id_minusl(p):
    "Factor : id minusl"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="sub\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"

#Globais
def p_Factor_gid_plus(p):
    "Factor : gid plus"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="add\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Factor_gid_plusl(p):
    "Factor : gid plusl"
```

```
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="add\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Factor_gid_minus(p):
    "Factor : gid minus"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="sub\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Factor_gid_minusl(p):
    "Factor : gid minusl"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="sub\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
```

Existem três tipos de operações básicas que podem ser aplicadas a um termo, multiplicação, divisão, e resto da divisão cada uma destas tem a sua própria instrução na máquina virtual `mul div mod`.

```
def p_Term_mul(p):
    "Term : Term '*' Factor"
    p[0] = p[1]+p[3]+"mul\n"

def p_Term_div(p):
    "Term : Term '/' Factor"
    p[0] = p[1]+p[3]+"div\n"

def p_Term_mod(p):
    "Term : Term '%' Factor"
    p[0] = p[1]+p[3]+"mod\n"

def p_Term_factor(p):
    "Term : Factor"
    p[0] = p[1]
```

De seguida temos um conjunto de instruções especiais para os termos, que guardam o valor final na variável id e podem deixar na stack o novo valor deste id ou o valor anterior.

```
#Multiplicação
```

```
def p_Term_id_muleq(p):
    "Term : id muleq Factor"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="mul\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"

def p_Term_id_muleql(p):
    "Term : id muleql Factor"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="mul\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"


#Divisão
def p_Term_id_diveq(p):
    "Term : id diveq Factor"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="div\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"

def p_Term_id_diveql(p):
    "Term : id diveql Factor"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="div\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"

#Resto da divisão

def p_Term_id_modeq(p):
    "Term : id modeq Factor"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="mod\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"

def p_Term_id_modeql(p):
    "Term : id modeql Factor"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
```

```
    p[0]+=p[3]
    p[0]+="mod\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"

#Globais

def p_Term_gid_muleq(p):
    "Term : gid muleq Factor"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="mul\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Term_gid_diveq(p):
    "Term : gid diveq Factor"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="div\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Term_gid_modeq(p):
    "Term : gid modeq Factor"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="mod\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"


def p_Term_gid_muleql(p):
    "Term : gid muleql Factor"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="mul\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Term_gid_diveql(p):
    "Term : gid diveql Factor"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="div\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Term_gid_modeql(p):
```

```
    "Term : gid modeql Factor"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="mod\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
```

Existem dois tipos de operações básicas que podem ser aplicadas a uma expressão, soma e subtração cada uma destas tem a sua própria instrução na máquina virtual `add` e `sub` respetivamente.

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

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

De seguida temos um conjunto de instruções especiais para as expressões, que guardam o valor final na variável id e podem deixar na stack o novo valor deste id ou o valor anterior.

```
#Soma

def p_Exp_id_addeq(p):
    "Exp : id addeq Term"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="add\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"

def p_Exp_id_addeql(p):
    "Exp : id addeql Term"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="add\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"

#Subtração

def p_Exp_id_subeq(p):
    "Exp : id subeq Term"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="sub\n"
```

```
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"

def p_Exp_id_subeql(p):
    "Exp : id subeql Term"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="sub\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"

#Globais

def p_Exp_gid_addeq(p):
    "Exp : gid addeq Term"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="add\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Exp_gid_subeq(p):
    "Exp : gid subeq Term"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="sub\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Exp_gid_addeql(p):
    "Exp : gid addeql Term"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="add\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Exp_gid_subeql(p):
    "Exp : gid subeql Term"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="sub\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"


def p_Exp_term(p):
    "Exp : Term"
```

```
    p[0] = p[1]
```

3.4.3 Atribuição de Variáveis

Para fazer as atribuições de variáveis, é necessário saber sempre se a variável a declarar é global ou não para saber que instrução da VM usar para aceder ao seu valor e que tabela usar para aceder seu offset offset.

Se for uma variável local usamos o `storel` em conjunto com com o valor a guardar e o offset da variável para guardar o valor na posição da stack desejada, se for um array começamos por usar `pushfp` em conjunto com `padd` e o offset da posição do array na stack para aceder á sua posição, por fim usando `storen` em conjunto com o indice do array a ser acedido e o valor a ser guardado para colocar o valor na posição correta na stack. Se for uma array bi dimensional apenas temos de calcular o indice a ser acedido multiplicando o número de colunas pelo índice da linha e somando finalmente o índice da coluna.

```
def p_Attr(p):
    "Attr : id '=' Exp"
    p[0]=p[3]
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"


def p_Attr_arr(p):
    "Attr : id '[' Exp ']' '=' Exp"
    p[0]="pushfp\n"
    p[0]+="pushi "+str(p.parser.table[p[1][0]])+"\n"
    p[0]+="padd\n"
    p[0]+=p[3]
    p[0]+=p[6]
    p[0]+="storen\n"


def p_Attr_arr2D(p):
    "Attr : id '[' Exp ']' '[' Exp ']' '=' Exp"
    p[0]="pushfp\n"
    p[0]+="pushi "+str(p.parser.table[p[1]][0])+"\n"
    p[0]+="padd\n"
    p[0]+=p[3]
    p[0]+="pushi "+str(p.parser.table[p[1]][1])+"\n" #Tamanho das linhas
    p[0]+="mul\n"
    p[0]+=p[6]
    p[0]+="add\n"
    p[0]+=p[9]
    p[0]+="storen\n"
```

Para a atribuição de variáveis globais são seguidos exatamente os mesmos passos, no entanto usamos o `storeg` e `pushgp` visto queremos somar o offset ao global pointer e não ao frame pointer.

```
def p_Attr_g(p):
    "Attr : gid '=' Exp"
    p[0]=p[3]
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"


def p_Attr_arrg(p):
    "Attr : gid '[' Exp ']' '=' Exp"
    p[0]="pushgp\n"
    p[0]+="pushi "+str(p.parser.tableG[p[1][1:]][0])+"\n"
    p[0]+="padd\n"
    p[0]+=p[3]
    p[0]+=p[6]
    p[0]+="storen\n"


def p_Attr_arr2Dg(p):
    "Attr : gid '[' Exp ']' '[' Exp ']' '=' Exp"

    p[0]="pushgp\n"
    p[0]+="pushi "+str(p.parser.tableG[p[1][1:]][0])+"\n"
    p[0]+="padd\n"
    p[0]+=p[3]
    p[0]+="pushi "+str(p.parser.tableG[p[1][1:]][1])+"\n" #Tamanho das linhas
    p[0]+="mul\n"
    p[0]+=p[6]
    p[0]+="add\n"
    p[0]+=p[9]
    p[0]+="storen\n"
```

3.4.4 Condições

A maior parte das instruções condicionas que queremos na nossa linguagem já está definida em assembly, maior, maior ou igual, menor, menor ou igual, inverso, igual correspondem a `sup`, `inf`, `supeq`, `infeq`, `not` e `equal`, respetivamente.

```
def p_Cond_sup(p):
    "Cond : Exp sup Exp"
    p[0] = p[1]+p[3]+"sup\n"

def p_Cond_inf(p):
    "Cond : Exp inf Exp"
    p[0] = p[1]+p[3]+"inf\n"

def p_Cond_supeq(p):
    "Cond : Exp supeq Exp"
```

```
    p[0] = p[1]+p[3]+"supeq\n"

def p_Cond_infeq(p):
    "Cond : Exp infeq Exp"
    p[0] = p[1]+p[3]+"infeq\n"

def p_Cond_not(p):
    "Cond : not Exp"
    p[0] = p[2]+"not\n"

def p_Cond_eq(p):
    "Cond : Exp eq Exp"
    p[0] = p[1]+p[3]+"equal\n"
```

Para fazer as restantes operações foi nessessário um conjunto adicional de instruções.
Para desigualdade simplesmente usamos as instruções `equal` e `not` em conjunto para obter o valor da sua igualdade mas deixando na stack o seu oposto.
De modo a fazer as instruções and e or, usamos as suas propriadades matemáticas, visto que uma condição tem o valor de 1(verdadeiro) ou 0(falso) podemos obter o resultado pretendido para operação `and` ao multiplicar os dois operadores um por outro, para a operação `or` é necessário fazer a subtração da adição dos operador com a multiplicação dos mesmos.

```
def p_Cond_diff(p):
    "Cond : Exp diff Exp"
    p[0] = p[1]+p[3]+"equal\n"+"not\n"

def p_Cond_and_par(p):
    "Cond : '(' Cond and Cond ')'"
    p[0] = p[2]+p[4]+"mul\n"

def p_Cond_or_par(p):
    "Cond : '(' Cond or Cond ')' "
    p[0] = p[2]+p[4]+"add\n"
    p[0]+=p[2]+p[4]+"mul\n"+"sub\n"
```

A parte mais importante para gerar o código assembly para as instruções if e else é manusear as flags, a partir da variável p.parser.currTag que é incrementada sempre que é criada uma nova label temos sempre a certeza que não vamos usar labels repetidas.
Um bloco com apenas um if necessita apenas de uma flag, começamos por escrever a condição e a instrução de salto (`jz`) que verifica se o último valor na stack é 0 fazendo o salto para a label que vai estar no fim do if se o valor for 0, depois adicionamos as instruções e por fim a flag
Por outro lado o bloco com um if e um else necessita de duas flags uma no fim do if e outra no fim do else, começamos então por colocar a condição inicial e depois um salto (`jz`) para a flag no fim do if, de seguida colocamos as instruções do primeiro bloco e um `jump` (salto incondicional) para o fim do else, por fim colocamos a flag de fim de if seguida das instruções do segundo bloco seguido da flag do fim do else.

```
def p_If(p):
    "If : if '(' Cond ')' '{' Insts '}' "
    p.parser.currTag+=1
    p[0]=p[3]+"jz ifEnd"+str(p.parser.currTag)+"\n"

    p[0]+=p[6]
    p[0]+="ifEnd"+str(p.parser.currTag)+":\n"


def p_If_Else(p):
    "If : if '(' Cond ')' '{' Insts '}' else '{' Insts '}'"
    p.parser.currTag+=1
    p[0]=p[3]+"jz ifEnd"+str(p.parser.currTag)+"\n"
    p[0]+=p[6]
    p[0]+="jump elseEnd"+str(p.parser.currTag)+"\n"
    p[0]+="ifEnd"+str(p.parser.currTag)+":\n"

    p[0]+=p[10]
    p[0]+="elseEnd"+str(p.parser.currTag)+":\n"
```

3.4.5 Instruções Cíclicas

Tal como nas intruções if e else a parte mais importante para conseguir representar as instruções ciclicas em código assembly é saber onde colocar as flags
Começando pelo ciclo for, temos de ter cuidado com a posição do conjunto de instruções no código assembly, começamos por colocar as instruções inicias e uma flag para o início do ciclo, depois colocamos a condição visto que esta é verificada sempre no início e um `jz` para o fim do ciclo, de seguida colocamos as restantes instruções tendo cuidado para colocar as que são referentes ao corpo do for primeiro e só depois as que são referentes ás instruções finais do cabeçalho do for visto que queremos executar estas no fim, finalmente a adicionamos um `jump` para o ínicio do ciclo e a flag de fim de ciclo.

```
def p_For(p):
    "For : for '(' Insts ';' Cond ';' Insts ')' '{' Insts '}'"
    p.parser.currTag+=1
    p[0]=p[3]
    p[0]+="forStart"+str(p.parser.currTag)+":\n"
    p[0]+=p[5]
    p[0]+="jz forEnd"+str(p.parser.currTag)+"\n"
    p[0]+=p[10]+p[7]
    p[0]+="jump forStart"+str(p.parser.currTag)+"\n"
    p[0]+="forEnd"+str(p.parser.currTag)+":\n"
```

O while é um bocado mais simples que o if não sendo necessário trocar a ordem das instruções, começamos por colocar a `flag` de inicio de ciclo, e a condição seguida de um `jz` para o fim do ciclo while, de seguida colocamos a as instruções do corpo do ciclo e um `jump` para o início do ciclo finalizando com a flag de fim de ciclo.

```
def p_While(p):
    "While : while '(' Cond ')' '{' Insts '}'"
    p.parser.currTag+=1
    p[0]="whileStart"+str(p.parser.currTag)+":\n"
    p[0]+=p[3]
    p[0]+="jz whileEnd"+str(p.parser.currTag)+"\n"
    p[0]+=p[6]
    p[0]+="jump whileStart"+str(p.parser.currTag)+"\n"
    p[0]+="whileEnd"+str(p.parser.currTag)+":\n"
```

Por fim temos o ciclo Repeat, na nossa versão este cíclo recebe uma expressão(n) e repete as instruções do corpo do ciclo n vezes, para conseguirmos executar este ciclo precisamos de tirar proveito de variaves reservadas para a sua execução, logo no início de cada função são inicializadas 10 variáveis a 0 que vão servir para este controlo.
Para saber o offset da variável da controlo necessária usamos a seguinte fórmula `(p.parser.level+1)%10`, isto dá nos sempre o próximo valor dísponivel para ser usado pelo ciclo, isto é necessário para permitir fazer o aninhamento de vários ciclos repeat, é de notar que visto que só temos 10 valores o número máximo de ciclos repeat aninhados é 10.
Tendo esta informação começamos por adicionar a flag inicial do ciclo repeat e as instruções do seu corpo, de seguida incrementamos o valor da variavel de controlo correspondente ao ciclo e verificamos se o novo valor desta variável é igual a 0, por fim saltamos para o inicio do ciclo e colocamos a variavel de controlo a zero.

```
def p_Repeat(p):
    "Repeat : repeat '(' Exp ')' '{' Insts '}' "
    p.parser.currTag+=1
    p.parser.level=(p.parser.level+1)%10
    addr = str(p.parser.level+p.parser.offset)

    p[0]="repeat"+str(p.parser.currTag)+":\n"

    p[0]+=p[6]
    p[0]+="pushl "+addr+"\n"
    p[0]+="pushi 1\n"
    p[0]+="add\n"
    p[0]+="storel "+addr+"\n"

    p[0]+="pushl "+addr+"\n"
    p[0]+=p[3]
    p[0]+="equal\n"
    p[0]+="jz repeat"+str(p.parser.currTag)+"\n"

    p[0]+="pushi 0\n"
```

```
    p[0]+="storel "+addr+"\n"
```

3.4.6 Funções Predefinidas

Para fazer o read começamos usar a instrução `read` que pede input ao utilizador e coloca a resposta na heap de strings sob a forma de string deixando na stack um apontador para essa string, esta instrução em conjunto com a instrução `atoi` que acede a uma string na heap através do seu apontador e transforma-a num inteiro deixando o resultado na stack, são suficientes para ler um número inteiro do standard input, por fim basta apenas guardar esse inteiro na variável escolhida pelo utilizador com a instrução `storel`

```
def p_Read(p):
    "Read : read '(' id ')'"

    p[0]="read\natoi\n"
    p[0]+="storel "+str(p.parser.table[p[3]])+"\n"
```

De seguida temos várias versões de funções que tratam de escrever uma string no standard output.
A função Print simplesmente escreve uma expressão no standard output com ajuda da instrução `writei` que retira um valor da stack e imprimeo no standard output.

```
def p_Print(p):
    "Print : print '(' Exp ')'"
    p[0]=p[3]
    p[0]+="writei\n"
```

A função Prints que coloca uma string na heap de strings com a instrução `pushs` e escreve-a no standard output a partir da instrução `writes` que dado um apontador para uma string na heap de strings coloca-a no standard output.

```
def p_Prints(p):
    "Prints : prints '(' string ')'"
    p[0]="pushs "+p[3]+"\n"
    p[0]+="writes\n"
```

Por fim temos a função Println faz a mesma coisa que Print mas também imprime um new line no fim da expressão com ajuda das instruções pushs e writes

```
def p_Println(p):
    "Println : println '(' Exp ')'"

    p[0]=p[3]
    p[0]+="writei\n"
    p[0]+="pushs \"\\n\"\n"
    p[0]+="writes\n"
```

3.4.7 Subprogramas

Os subprogramas são definidos depois da função main, começam com a sua identificação que é colocada no ficheiro de saída como se fosse uma flag, de seguida são colocadas as declarações de variáveis e as instruções do subprograma por fim é feito um return de zero apartir do `storel` -1 que coloca esse valor na posição anterior ao frame pointer atual (esta está sempre livre porque sempre que é chamada uma função é feito um `push` de 0), seguidamente é feito um `return` que retira da call stack o apontador para o código(fp) e o apontador para a frame (pc) do estado anterior restaurando-o, e por fim as variáveis de controlo de estado são postas no seu estado inicial.
É de notar que apesar de escrevermos este return no ficheiro sáida este nem sempre vai ser executado visto que o utilizador pode introduzir o seu próprio return como vamos ver de seguida.

```
def p_DefBlc(p):
    "DefBlc : id '{' VarBlcs Insts '}' "
    p[0]=p[1]+":\n"
    p[0]+=p[3]+p[4]
    p[0]+="pushi 0\nstorel -1\nreturn\n"

    parser.table = {}
    parser.offset = 0
    parser.level = -1
```

Para chamar uma função é necessário começar por fazer `push` de uma variável que vai ser usada para armazenar o resultado da função chamada, depois basta fazer `pusha` do id da função e `call`, estas funções tratam de guardar o pc e fp atual, e atualizam o novo fp e pc da função com a label id chamada.

```
def p_Factor_call(p):
    "Factor : id '(' ')' "
    p[0] = "pushi 0\n"
    p[0]+= "pusha "+str(p[1])+"\ncall\n"
```

Por fim temos a instrução return, durante a execução de um subprograma o utilizador tem a possibilidade de retornar um valor para o programa anterior, para o fazer começamos por colocar a expressão a retornar na stack e a partir da instrução `storel` -1 colocamos esse valor na posição anterior ao frame pointer atual, visto que esta posição está livre porque foi feito um `pushi 0` antes de ser chamada a função, por fim fazemos return para voltar ao estado anterior.

```
def p_Return(p):
    "Return : return '(' Exp ')'"
    p[0]=p[3]
    p[0]+="storel -1\n"
    p[0]+="return\n"
```

3.4.8 Programa Total

Um programa começa, opcionalmente, com a declaração de um conjunto de variáveis globais.
É necessário antes de se declarar qualquer variável avisar que vão ser declaradas variáveis globais e não locais, por isso foi criada a produção GlobalBegin com este objetivo.

No fim de serem declaradas as diversas variáveis é colocado o offset a 0 é indicado que não vão ser declaradas mais variáveis globais.

```
def p_GlobalBlc(p):
    "GlobalBlc : GlobalBegin VarBlcs '}' "
    p[0]=p[1]+p[2]
    parser.offset = 0
    p.parser.isGlobal=False

def p_GlobalBlc_Null(p):
    "GlobalBlc : "
    p[0]=""

def p_GlobalBegin(p):
    "GlobalBegin : global '{' "
    p.parser.isGlobal=True
    p[0]=""
```

De seguida é declarada a função main, que também tem um bloco de variáveis e um conjunto de instruções. É necessário usar a instrução `start` para coloca o frame pointer que estava a zero na posição do stack pointer atual, isto é necessário visto que podem ter sido declaradas variáveis globais antes da função, depois são colocadas a declarações de variáveis e é feito um `pushn 10` para as variáveis de controlo dos ciclos repeat, por fim são colocadas as instruções, o programa é parado com a instrução assembly `stop` e as variáveis de controlo de estado são postas no seu estado inicial para estarem prontas a ser usadas se ouverem declarações de subprogramas.

```
def p_MainBlc(p):
    "MainBlc : main '{' VarBlcs Insts '}'"
    p[0]="start\n"
    p[0]+=p[3]+"pushn 10\n"
    p[0]+=p[4]+"stop\n"

    parser.table = {}
    parser.offset = 0
    parser.level = -1
```

Finalmente depois de todo o programa ser traduzido para código assembly é escrito no ficheiro de saida.

```
def p_Prog(p):
    "Prog :  GlobalBlc MainBlc DefBlcs"
    out.write(p[1]+p[2]+p[3])
```

Ambos os ficheiros de saída e entrada são obtidos com o apoio do módulo sys de python

```
parser = yacc.yacc()

f = open(sys.argv[1], "r")
out = open(sys.argv[2], "w")

result = parser.parse(f.read())
```

Capítulo 4

Codificação e Testes

Mostram-se a seguir alguns testes feitos (valores introduzidos) e os respectivos resultados obtidos

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

4.1.1 Código neoC

```
main {
    int{
        start
        end
        r
    }

    prints("Introduza número inicial:\n")
    read(start)
    prints("Introduza número final:\n")
    read(end)

    for (r=0 ; start<=end ; start++){
        if ((start%2)!=0){
            prints("Impar: ")
            println(start)
            r++
        }
    }

    prints("Total de números ímpares: ")
    println(r)

}
```

4.1.2 Código assembly

```
start
pushi 0
pushi 0
pushi 0
pushn 10
pushs "Introduza número inicial:\n"
writes
read
atoi
storel 0
pushs "Introduza número final:\n"
writes
read
atoi
storel 1
pushi 0
storel 2
forStart2:
pushl 0
pushl 1
infeq
jz forEnd2
pushl 0
pushi 2
mod
pushi 0
equal
not
jz ifEnd1
pushs "Impar: "
writes
pushl 0
writei
pushs "\n"
writes
pushl 2
pushi 1
add
storel 2
pushl 2
ifEnd1:
pushl 0
pushi 1
add
storel 0
pushl 0
```

```
jump forStart2
forEnd2:
pushs "Total de números ímpares: "
writes
pushl 2
writei
pushs "\n"
writes
stop
```

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

4.2.1 Código neoC

```
main {
    int{
        inp[100]
        n
        i
        aux
    }

    prints("Quantos números deseja introduzir? (máximo 100)\n")
    read(n)

    prints("Introduza ") print(n) prints(" números:\n")
    for(i=0;i<n;i++){
        read(aux)
        inp[i]=aux
    }
    prints("Ordem Inversa: \n")
    for(i=n-1;i>=0;i--){
        println(inp[i])
    }

}
```

4.2.2 Código assembly

```
start
pushn 100
pushi 0
pushi 0
pushi 0
pushn 10
pushs "Quantos números deseja introduzir? (máximo 100)\n"
```

```
writes
read
atoi
storel 100
pushs "Introduza "
writes
pushl 100
writei
pushs " números:\n"
writes
pushi 0
storel 101
forStart1:
pushl 101
pushl 100
inf
jz forEnd1
read
atoi
storel 102
pushfp
pushi 0
padd
pushl 101
pushl 102
storen
pushl 101
pushi 1
add
storel 101
pushl 101
jump forStart1
forEnd1:
pushs "Ordem Inversa: \n"
writes
pushl 100
pushi 1
sub
storel 101
forStart2:
pushl 101
pushi 0
supeq
jz forEnd2
pushfp
pushi 0
padd
pushl 101
```

```
loadn
writei
pushs "\n"
writes
pushl 101
pushi 1
sub
storel 101
pushl 101
jump forStart2
forEnd2:
stop
```

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

4.3.1 Código neoC

```
main {
    int {
        n
        i
        in
        menor
    }
    prints("Quantos números deseja ler:\n")
    read(n)
    prints("Insira um número:\n")
    read(menor)
    for(i=1;i<n;i=i+1){
        prints("Insira um número:\n")
        read(in)
        if (in<menor) { menor=in }
    }

    prints("Menor número inserido: ")
    println(menor)
}
```

4.3.2 Código assembly

```
start
pushi 0
pushi 0
pushi 0
pushi 0
pushn 10
pushs "Quantos números deseja ler:\n"
```

```
writes
read
atoi
storel 0
pushs "Insira um número:\n"
writes
read
atoi
storel 3
pushi 1
storel 1
forStart2:
pushl 1
pushl 0
inf
jz forEnd2
pushs "Insira um número:\n"
writes
read
atoi
storel 2
pushl 2
pushl 3
inf
jz ifEnd1
pushl 2
storel 3
ifEnd1:
pushl 1
pushi 1
add
storel 1
jump forStart2
forEnd2:
pushs "Menor número inserido: "
writes
pushl 3
writei
pushs "\n"
writes
stop
```

4.4 Invocar e usar num programa seu uma função 'potencia()', que começa por ler do input a base B e o expoente E e retorna o valor B^E

4.4.1 Código neoC

```
main {

    println(pow()+pow())
}

pow {
    int{
        base
        exp
        i
        r
    }

    prints("Insira base:\n")
    read(base)

    prints("Insira expoente:\n")
    read(exp)

    for(i=0 r=1;i<exp;i++){
        r*=base
    }

    return(r)
}
```

4.4.2 Código assembly

```
start
pushn 10
pushi 0
pusha pow
call
pushi 0
pusha pow
call
add
writei
pushs "\n"
writes
stop
pow:
```

```
pushi 0
pushi 0
pushi 0
pushi 0
pushs "Insira base:\n"
writes
read
atoi
storel 0
pushs "Insira expoente:\n"
writes
read
atoi
storel 1
pushi 0
storel 2
pushi 1
storel 3
forStart1:
pushl 2
pushl 1
inf
jz forEnd1
pushl 3
pushl 0
mul
storel 3
pushl 3
pushl 2
pushi 1
add
storel 2
pushl 2
jump forStart1
forEnd1:
pushl 3
storel -1
return
pushi 0
storel -1
return
```

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

4.5.1 Código neoC

```
    main {
    int {
        n
        r=1
    }

    prints("Introduza numero (0 para terminar):\n")
    read(n)
    while (n!=0){
        r*=n
        prints("Introduza numero (0 para terminar):\n")
        read(n)
    }
    prints("Produtorio: \n")
    println(r)
}
```

4.5.2 Código assembly

```
start
pushi 0
pushi 1
pushn 10
pushs "Introduza numero (0 para terminar):\n"
writes
read
atoi
storel 0
whileStart1:
pushl 0
pushi 0
equal
not
jz whileEnd1
pushl 1
pushl 0
mul
storel 1
pushl 1
pushs "Introduza numero (0 para terminar):\n"
writes
read
atoi
```

```
storel 0
jump whileStart1
whileEnd1:
pushs "Produtorio: \n"
writes
pushl 1
writei
pushs "\n"
writes
stop
```

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

4.6.1 Código neoC

```
    main {
    int {
        r
        init
        inp
        i
    }

    prints("Insira 4 números:\n")
    read(init)
    inp=init

    for (i=0;i<3;i++){
        read(inp)
        r+=(inp==init)
    }
    if (r==3){
        prints("Números introduzidos podem ser lados de um quadrado\n")
    } else {
        prints("Números introduzidos não podem ser lados de um quadrado\n")
    }
}
```

4.6.2 Código assembly

```
start
pushi 0
pushi 0
pushi 0
pushi 0
pushn 10
pushs "Insira 4 números:\n"
writes
```

```
read
atoi
storel 1
pushl 1
storel 2
pushi 0
storel 3
forStart1:
pushl 3
pushi 3
inf
jz forEnd1
read
atoi
storel 2
pushl 0
pushl 2
pushl 1
equal
add
storel 0
pushl 0
pushl 3
pushi 1
add
storel 3
pushl 3
jump forStart1
forEnd1:
pushl 0
pushi 3
equal
jz ifEnd2
pushs "Números introduzidos podem ser lados de um quadrado\n"
writes
jump elseEnd2
ifEnd2:
pushs "Números introduzidos não podem ser lados de um quadrado\n"
writes
elseEnd2:
stop
```

Capítulo 5

Conclusão

A solução de software encontrado gera um compilador que permite declarar variáveis, fazer a atribuição de valores a certas variáveis, ler e escrever no standard input e output e efetuar instruções cíclicas e condicionais.

Apêndice A

Código do Programa

A.1 Analizador Léxico

Lista-se a seguir o código do analizador léxico do programa compiler_lex.py que foi desenvolvido.

```
import ply.lex as lex

tokens = ['num', 'id', 'int','print','println','prints','string','main','repeat','read','sup','in
,'infeq','not','diff','and','or','if','else','for','while','plus','addeq','subeq','diveq','muleq
,'muleql','minus','plusl','minusl','modeq','modeql','return','global','gid']
literals = ['=','(',')','{','}','+','*','-','/','[',']',';','%',',']

def t_numR(t):
    r'\d+\.\d+'
    t.value = float(t.value)
    return t

def t_num(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_string(t):
    r'"[^"]*"'
    return t

def t_return(t):
    r'return'
    return t

def t_global(t):
    r'global'
    return t

def t_println(t):
    r'println'
```

```
    return t

def t_read(t):
    r'read'
    return t

def t_prints(t):
    r'prints'
    return t

def t_print(t):
    r'print'
    return t

def t_int(t):
    r'int'
    return t

def t_for(t):
    r'for'
    return t

def t_while(t):
    r'while'
    return t

def t_vars(t):
    r'vars'
    return t

def t_repeat(t):
    r'repeat'
    return t

def t_main(t):
    r'main'
    return t

def t_if(t):
    r'if'
    return t

def t_else(t):
    r'else'
    return t

def t_diveql(t):
    r'\./='
```

```
    return t

def t_addeql(t):
    r'\.\+\='
    return t

def t_subeql(t):
    r'\.\-\='
    return t


def t_muleql(t):
    r'\.\*\='
    return t

def t_modeql(t):
    r'\.\%\='
    return t

def t_plus(t):
    r'\+\+'
    return t

def t_plusl(t):
    r'\.\+'
    return t

def t_minus(t):
    r'--'
    return t

def t_minusl(t):
    r'\.-'
    return t

def t_addeq(t):
    r'\+\='
    return t

def t_subeq(t):
    r'\-\='
    return t

def t_diveq(t):
    r'/='
    return t

def t_muleq(t):
```

```
    r'\*\='
    return t

def t_modeq(t):
    r'\%\='
    return t


def t_supeq(t):
    r'>='
    return t

def t_infeq(t):
    r'<='
    return t

def t_sup(t):
    r'>'
    return t

def t_inf(t):
    r'<'
    return t

def t_diff(t):
    r'!='
    return t

def t_not(t):
    r'!'
    return t

def t_eq(t):
    r'=='
    return t

def t_and(t):
    r'&&'
    return t

def t_or(t):
    r'\|\|'
    return t

def t_gid(t):
    r'&[a-zA-Z]\w*'
    return t
```

```
def t_id(t):
    r'[a-zA-Z]\w*'
    return t

t_ignore = " \t\n"

# Tratamento de erros
def t_error(t):
    print('Caracter ilegal:', t.value[0])
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()
```

Lista-se a seguir UM TEXTO (COM O COMANDO VERBATIN)

```
aqui deve aparecer o código do programa,
tal como está formato no ficheiro-fonte "darius.java"
um pouco de matematica $\$$
caso indesejável $\varepsilon$
```

A.2 Gramática Tradutora

Lista-se a seguir o código do analizador léxico do programa compiler_yacc.py que foi desenvolvido.

```
import sys
import ply.yacc as yacc
from compiler_lex import tokens

def p_Prog(p):
    "Prog :  GlobalBlc MainBlc DefBlcs"
    out.write(p[1]+p[2]+p[3])

def p_DefBlcs(p):
    "DefBlcs : DefBlcs DefBlc "
    p[0] = p[1]+p[2]

def p_DefBlcs_End(p):
    "DefBlcs : "
    p[0] = ""

def p_DefBlc(p):
    "DefBlc : id '{' VarBlcs Insts '}' "
    p[0]=p[1]+":\n"
    p[0]+=p[3]+p[4]
```

```
    p[0]+="pushi 0\nstorel -1\nreturn\n"

    parser.table = {}
    parser.offset = 0
    parser.level = -1




def p_GlobalBlc(p):
    "GlobalBlc : GlobalBegin VarBlcs '}' "
    p[0]=p[1]+p[2]
    parser.offset = 0
    p.parser.isGlobal=False

def p_GlobalBegin(p):
    "GlobalBegin : global '{' "
    p.parser.isGlobal=True
    p[0]=""

def p_GlobalBlc_Null(p):
    "GlobalBlc : "
    p[0]=""

def p_VarBlcs(p):
    "VarBlcs : VarBlcs BlcInt "
    p[0]=p[1]+p[2]

def p_VarBlcs_End(p):
    "VarBlcs : "
    p[0]=""

def p_BlcInt(p):
    "BlcInt : int '{' Dcls '}' "
    p[0]=p[3]


def p_Dcls(p):
    "Dcls : Dcls Dcl"
    p[0]=p[1]+p[2]

def p_Dcls_End(p):
    "Dcls : "
    p[0]=""


def p_MainBlc(p):
    "MainBlc : main '{' VarBlcs Insts '}'"
```

```
    p[0]="start\n"
    p[0]+=p[3]+"pushn 10\n"
    p[0]+=p[4]+"stop\n"

    parser.table = {}
    parser.offset = 0
    parser.level = -1

def p_Insts(p):
    "Insts : Insts Inst "
    p[0]=p[1]+p[2]

def p_Insts_End(p):
    "Insts :"
    p[0]=""

def p_Inst_Attr(p):
    "Inst : Attr"
    p[0]=p[1]

def p_Inst_Return(p):
    "Inst : Return"
    p[0]=p[1]

def p_Inst_Exp(p):
    "Inst : Exp"
    p[0]=p[1]

def p_Inst_Print(p):
    "Inst : Print"
    p[0]=p[1]

def p_Inst_Println(p):
    "Inst : Println"
    p[0]=p[1]

def p_Inst_Prints(p):
    "Inst : Prints"
    p[0]=p[1]


def p_Inst_Repeat(p):
    "Inst : Repeat"
    p[0]=p[1]

def p_Inst_For(p):
    "Inst : For"
    p[0]=p[1]
```

```
def p_Inst_While(p):
    "Inst : While"
    p[0]=p[1]

def p_Inst_Read(p):
    "Inst : Read"
    p[0]=p[1]


def p_Inst_If(p):
    "Inst : If"
    p[0]=p[1]




def p_Repeat(p):
    "Repeat : repeat '(' Exp ')' '{' Insts '}' "
    p.parser.currTag+=1
    p.parser.level=(p.parser.level+1)%10
    p[0]="repeat"+str(p.parser.currTag)+":\n"

    p[0]+=p[6]
    addr = str(p.parser.level+p.parser.offset)
    p[0]+="pushl "+addr+"\n"
    p[0]+="pushi 1\n"
    p[0]+="add\n"
    p[0]+="storel "+addr+"\n"

    p[0]+="pushl "+addr+"\n"
    p[0]+=p[3]
    p[0]+="equal\n"
    p[0]+="jz repeat"+str(p.parser.currTag)+"\n"

    p[0]+="pushi 0\n"
    p[0]+="storel "+addr+"\n"


def p_For(p):
    "For : for '(' Insts ';' Cond ';' Insts ')' '{' Insts '}'"
    p.parser.currTag+=1
    p[0]=p[3]
    p[0]+="forStart"+str(p.parser.currTag)+":\n"
    p[0]+=p[5]
    p[0]+="jz forEnd"+str(p.parser.currTag)+"\n"
    p[0]+=p[10]+p[7]
    p[0]+="jump forStart"+str(p.parser.currTag)+"\n"
```

```
    p[0]+="forEnd"+str(p.parser.currTag)+":\n"

def p_While(p):
    "While : while '(' Cond ')' '{' Insts '}'"
    p.parser.currTag+=1
    p[0]="whileStart"+str(p.parser.currTag)+":\n"
    p[0]+=p[3]
    p[0]+="jz whileEnd"+str(p.parser.currTag)+"\n"
    p[0]+=p[6]
    p[0]+="jump whileStart"+str(p.parser.currTag)+"\n"
    p[0]+="whileEnd"+str(p.parser.currTag)+":\n"


def p_Read(p):
    "Read : read '(' id ')'"

    p[0]="read\natoi\n"
    p[0]+="storel "+str(p.parser.table[p[3]])+"\n"

def p_Print(p):
    "Print : print '(' Exp ')'"
    p[0]=p[3]
    p[0]+="writei\n"

def p_Println(p):
    "Println : println '(' Exp ')'"

    p[0]=p[3]
    p[0]+="writei\n"
    p[0]+="pushs \"\\n\"\n"
    p[0]+="writes\n"

def p_Prints(p):
    "Prints : prints '(' string ')'"
    p[0]="pushs "+p[3]+"\n"
    p[0]+="writes\n"

def p_Dcl_Arr(p):
    "Dcl : id '[' num ']'"
    p[0]="pushn "+str(p[3])+"\n"
    if p.parser.isGlobal:
        p.parser.tableG[p[1]]=(p.parser.offset,0)
    else:
        p.parser.table[p[1]]=(p.parser.offset,0)
    p.parser.offset+=p[3]

def p_Dcl_Arr_valEq(p):
    "Dcl : id '[' num ']' '=' num"
```

```
    p[0]=""
    for i in range(p[3]):
        p[0]+="pushi "+str(p[6])+"\n"

    if p.parser.isGlobal:
        p.parser.tableG[p[1]]=(p.parser.offset,0)
    else:
        p.parser.table[p[1]]=(p.parser.offset,0)
    p.parser.offset+=p[3]

def p_Dcl_Arr_val(p):
    "Dcl : id '[' ']' '=' '{' Nums '}'"
    p[0]=p[6]
    if p.parser.isGlobal:
        p.parser.tableG[p[1]]=(p.parser.offset,0)
    else:
        p.parser.table[p[1]]=(p.parser.offset,0)
    p.parser.offset+=p.parser.arraySize
    p.parser.arraySize=0

def p_Nums(p):
    "Nums : Nums num"
    p.parser.arraySize+=1
    p[0]=p[1]+"pushi "+str(p[2])+"\n"

def p_Nums_End(p):
    "Nums : "
    p[0]= ""

def p_Dcl_Arr_2D_valEq(p):
    "Dcl : id '[' num ']' '[' num ']' '=' num"
    p[0]=""
    for i in range(p[3]*p[6]):
        p[0]+="pushi "+str(p[9])+"\n"

    if p.parser.isGlobal:
        p.parser.tableG[p[1][1:]]=(p.parser.offset,p[6])
    else:
        p.parser.table[p[1]]=(p.parser.offset,p[6])
    p.parser.offset+=p[3]*p[6]

def p_Dcl_Arr_2D(p):
    "Dcl : id '[' num ']' '[' num ']'"
    p[0]="pushn "+str(p[3]*p[6])+"\n"
    if p.parser.isGlobal:
        p.parser.tableG[p[1][1:]]=(p.parser.offset,p[6])
    else:
        p.parser.table[p[1]]=(p.parser.offset,p[6])
```

```
    p.parser.offset+=p[3]*p[6]

def p_Dcl_Arr_2D_val(p):
    "Dcl : id '[' ']' '[' ']' '=' '{' BlcsNums '}' "
    p[0]=p[8]
    if p.parser.isGlobal:
        p.parser.tableG[p[1][1:]]=(p.parser.offset,int(p.parser.arraySize/p.parser.linhas))
    else:
        p.parser.table[p[1]]=(p.parser.offset,int(p.parser.arraySize/p.parser.linhas))
    p.parser.offset+=p.parser.arraySize
    p.parser.arraySize=0
    p.parser.linhas=0

def p_BlcsNums(p):
    "BlcsNums : BlcsNums '{' Nums '}'"
    p.parser.linhas+=1
    p[0]=p[1]+p[3]

def p_BlcsNums_End(p):
    "BlcsNums : "
    p[0]=""

def p_Dcl_0(p):
    "Dcl : id"
    p[0]=("pushi 0\n")
    if p.parser.isGlobal:
        p.parser.tableG[p[1]]=p.parser.offset
    else:
        p.parser.table[p[1]]=p.parser.offset
    p.parser.offset+=1

def p_Dcl_num(p):
    "Dcl : id '=' num"
    p[0]="pushi "+str(p[3])+"\n"
    if p.parser.isGlobal:
        p.parser.tableG[p[1]]=p.parser.offset
    else:
        p.parser.table[p[1]]=p.parser.offset
    p.parser.offset+=1



def p_Attr(p):
    "Attr : id '=' Exp"

    p[0]=p[3]
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
```

```
def p_Attr_arr(p):
    "Attr : id '[' Exp ']' '=' Exp"
    p[0]="pushfp\n"
    p[0]+="pushi "+str(p.parser.table[p[1]][0])+"\n"
    p[0]+="padd\n"
    p[0]+=p[3]
    p[0]+=p[6]
    p[0]+="storen\n"


def p_Attr_arr2D(p):
    "Attr : id '[' Exp ']' '[' Exp ']' '=' Exp"

    p[0]="pushfp\n"
    p[0]+="pushi "+str(p.parser.table[p[1]][0])+"\n"
    p[0]+="padd\n"
    p[0]+=p[3]
    p[0]+="pushi "+str(p.parser.table[p[1]][1])+"\n" #Tamanho das linhas
    p[0]+="mul\n"
    p[0]+=p[6]
    p[0]+="add\n"
    p[0]+=p[9]
    p[0]+="storen\n"

def p_Attr_g(p):
    "Attr : gid '=' Exp"

    p[0]=p[3]
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"


def p_Attr_arrg(p):
    "Attr : gid '[' Exp ']' '=' Exp"

    p[0]="pushgp\n"
    p[0]+="pushi "+str(p.parser.tableG[p[1][1:]][0])+"\n"
    p[0]+="padd\n"
    p[0]+=p[3]
    p[0]+=p[6]
    p[0]+="storen\n"


def p_Attr_arr2Dg(p):
    "Attr : gid '[' Exp ']' '[' Exp ']' '=' Exp"

    p[0]="pushgp\n"
    p[0]+="pushi "+str(p.parser.tableG[p[1][1:]][0])+"\n"
```

```
    p[0]+="padd\n"
    p[0]+=p[3]
    p[0]+="pushi "+str(p.parser.tableG[p[1][1:]][1])+"\n" #Tamanho das linhas
    p[0]+="mul\n"
    p[0]+=p[6]
    p[0]+="add\n"
    p[0]+=p[9]
    p[0]+="storen\n"

def p_Return(p):
    "Return : return '(' Exp ')'"
    p[0]=p[3]
    p[0]+="storel -1\n"
    p[0]+="return\n"

def p_If(p):
    "If : if '(' Cond ')' '{' Insts '}' "
    p.parser.currTag+=1
    p[0]=p[3]+"jz ifEnd"+str(p.parser.currTag)+"\n"

    p[0]+=p[6]
    p[0]+="ifEnd"+str(p.parser.currTag)+":\n"



def p_If_Else(p):
    "If : if '(' Cond ')' '{' Insts '}' else '{' Insts '}'"
    p.parser.currTag+=1
    p[0]=p[3]+"jz ifEnd"+str(p.parser.currTag)+"\n"
    p[0]+=p[6]
    p[0]+="jump elseEnd"+str(p.parser.currTag)+"\n"
    p[0]+="ifEnd"+str(p.parser.currTag)+":\n"

    p[0]+=p[10]
    p[0]+="elseEnd"+str(p.parser.currTag)+":\n"




def p_Exp_add(p):
    "Exp : Exp '+' Term"
    p[0] = p[1]+p[3]+"add\n"

def p_Exp_sub(p):
    "Exp : Exp '-' Term"
    p[0] = p[1]+p[3]+"sub\n"

def p_Exp_id_addeq(p):
```

```
    "Exp : id addeq Term"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="add\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"

def p_Exp_id_subeq(p):
    "Exp : id subeq Term"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="sub\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"

def p_Exp_id_addeql(p):
    "Exp : id addeql Term"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="add\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"

def p_Exp_id_subeql(p):
    "Exp : id subeql Term"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="sub\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"

def p_Exp_gid_addeq(p):
    "Exp : gid addeq Term"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="add\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Exp_gid_subeq(p):
    "Exp : gid subeq Term"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="sub\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Exp_gid_addeql(p):
```

```
    "Exp : gid addeql Term"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="add\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Exp_gid_subeql(p):
    "Exp : gid subeql Term"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="sub\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"


def p_Exp_term(p):
    "Exp : Term"
    p[0] = p[1]

def p_Term_mul(p):
    "Term : Term '*' Factor"
    p[0] = p[1]+p[3]+"mul\n"

def p_Term_div(p):
    "Term : Term '/' Factor"
    p[0] = p[1]+p[3]+"div\n"

def p_Term_mod(p):
    "Term : Term '%' Factor"
    p[0] = p[1]+p[3]+"mod\n"

def p_Term_id_muleq(p):
    "Term : id muleq Factor"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="mul\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"

def p_Term_id_diveq(p):
    "Term : id diveq Factor"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="div\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
```

```
def p_Term_id_modeq(p):
    "Term : id modeq Factor"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="mod\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"


def p_Term_id_muleql(p):
    "Term : id muleql Factor"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="mul\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"

def p_Term_id_diveql(p):
    "Term : id diveql Factor"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="div\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"

def p_Term_id_modeql(p):
    "Term : id modeql Factor"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+=p[3]
    p[0]+="mod\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"

def p_Term_gid_muleq(p):
    "Term : gid muleq Factor"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="mul\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Term_gid_diveq(p):
    "Term : gid diveq Factor"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="div\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
```

```
def p_Term_gid_modeq(p):
    "Term : gid modeq Factor"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="mod\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"


def p_Term_gid_muleql(p):
    "Term : gid muleql Factor"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="mul\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Term_gid_diveql(p):
    "Term : gid diveql Factor"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="div\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Term_gid_modeql(p):
    "Term : gid modeql Factor"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+=p[3]
    p[0]+="mod\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Term_factor(p):
    "Term : Factor"
    p[0] = p[1]

def p_Factor_id_plus(p):
    "Factor : id plus"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="add\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"

def p_Factor_id_plusl(p):
    "Factor : id plusl"
```

```
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="add\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"

def p_Factor_id_minus(p):
    "Factor : id minus"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="sub\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"

def p_Factor_id_minusl(p):
    "Factor : id minusl"
    p[0]="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushl "+str(p.parser.table[p[1]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="sub\n"
    p[0]+="storel "+str(p.parser.table[p[1]])+"\n"

def p_Factor_gid_plus(p):
    "Factor : gid plus"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="add\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Factor_gid_plusl(p):
    "Factor : gid plusl"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="add\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Factor_gid_minus(p):
    "Factor : gid minus"
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="sub\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Factor_gid_minusl(p):
    "Factor : gid minusl"
```

```
    p[0]="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushg "+str(p.parser.tableG[p[1][1:]])+"\n"
    p[0]+="pushi 1\n"
    p[0]+="sub\n"
    p[0]+="storeg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Factor_id(p):
    "Factor : id"
    p[0] = "pushl "+str(p.parser.table[p[1]])+"\n"

def p_Factor_gid(p):
    "Factor : gid"
    p[0] = "pushg "+str(p.parser.tableG[p[1][1:]])+"\n"

def p_Factor_num(p):
    "Factor : num"
    p[0] = "pushi "+str(p[1])+"\n"

def p_Factor_call(p):
    "Factor : id '(' ')' "
    p[0] = "pushi 0\n"
    p[0]+= "pusha "+str(p[1])+"\ncall\n"


def p_Factor_cond(p):
    "Factor : '(' Cond ')'"
    p[0] = p[2]


def p_Factor_arr(p):
    "Factor : id '[' Exp ']'"
    p[0]="pushfp\n"
    p[0]+="pushi "+str(p.parser.table[p[1]][0])+"\n"
    p[0]+="padd\n"+p[3]
    p[0]+="loadn\n"

def p_Factor_arr_2d(p):
    "Factor : id '[' Exp ']' '[' Exp ']'"
    p[0]="pushfp\n"
    p[0]+="pushi "+str(p.parser.table[p[1]][0])+"\n"
    p[0]+="padd\n"+str(p[3])
    p[0]+="pushi "+str(p.parser.table[p[1]][1])+"\n"
    p[0]+="mul\n"+str(p[6])
    p[0]+="add\n"+"loadn\n"

def p_Factor_arrg(p):
    "Factor : gid '[' Exp ']'"
    p[0]="pushgp\n"
```

```
    p[0]+="pushi "+str(p.parser.tableG[p[1][1:]][0])+"\n"
    p[0]+="padd\n"+p[3]
    p[0]+="loadn\n"

def p_Factor_arrg_2d(p):
    "Factor : gid '[' Exp ']' '[' Exp ']'"
    p[0]="pushgp\n"
    p[0]+="pushi "+str(p.parser.tableG[p[1][1:]][0])+"\n"
    p[0]+="padd\n"+str(p[3])
    p[0]+="pushi "+str(p.parser.tableG[p[1][1:]][1])+"\n"
    p[0]+="mul\n"+str(p[6])
    p[0]+="add\n"+"loadn\n"


def p_Factor_group(p):
    "Factor : '(' Exp ')'"
    p[0] = p[2]


def p_Cond_and_par(p):
    "Cond : '(' Cond and Cond ')'"
    p[0] = p[2]+p[4]+"mul\n"

def p_Cond_or_par(p):
    "Cond : '(' Cond or Cond ')' "
    p[0] = p[2]+p[4]+"add\n"
    p[0]+=p[2]+p[4]+"mul\n"+"sub\n"

def p_Cond_sup(p):
    "Cond : Exp sup Exp"
    p[0] = p[1]+p[3]+"sup\n"

def p_Cond_inf(p):
    "Cond : Exp inf Exp"
    p[0] = p[1]+p[3]+"inf\n"

def p_Cond_supeq(p):
    "Cond : Exp supeq Exp"
    p[0] = p[1]+p[3]+"supeq\n"

def p_Cond_infeq(p):
    "Cond : Exp infeq Exp"
    p[0] = p[1]+p[3]+"infeq\n"

def p_Cond_not(p):
    "Cond : not Exp"
    p[0] = p[2]+"not\n"
```

```
def p_Cond_eq(p):
    "Cond : Exp eq Exp"
    p[0] = p[1]+p[3]+"equal\n"

def p_Cond_diff(p):
    "Cond : Exp diff Exp"
    p[0] = p[1]+p[3]+"equal\n"+"not\n"


def p_error(p):
    print('Syntax error: ', p)

parser = yacc.yacc()
parser.tableG = {}
parser.table = {}
parser.offset = 0
parser.level = -1
parser.currTag = 0
parser.isGlobal = False
parser.arraySize = 0
parser.linhas = 0

f = open(sys.argv[1], "r")
out = open(sys.argv[2], "w")

result = parser.parse(f.read())
```