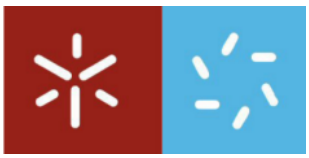

Licenciatura em Ciências da Computação

Processamento de Linguagens e Compiladores

(Grupo-6)

Trabalho Prático 2

Relatório de Desenvolvimento

Bruno Miguel Fernandes Araújo (a97509)

Bruna Micaela Rodrigues Araújo (a84914)

Filipe José Silva Castro (a96156)

15 Janeiro 2023

Conteúdo

Capítulo 1

Enunciado

Enunciado que nos foi fornecido:

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 de seleção para controlo do fluxo de execução.

• efetuar instruções de repetição (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.

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.

Como é da praxe neste tipo de linguagens, as variáveis deverão ser declaradas no início do programa e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explicitado, o valor da variável após a declaração é 0 (zero).

Desenvolva, então, um compilador para essa linguagem com base na GIC criada acima e com recurso aos módulos Yacc/ Lex do PLY/Python.
O compilador deve gerar pseudo-código, Assembly da Máquina Virtual VM.

Muito Importante: Para a entrega do TP deve preparar um conjunto de testes (programas-fonte escritos na sua linguagem) e mostrar o código Assembly gerado bem como o programa a correr na máquina virtual VM.

Capítulo 2

Introdução

No âmbito da cadeira de Processamento de Linguagens e Compiladores foi-nos proposto pelo educando um trabalho que tem como objetivo desenvolvermos uma linguagem de programação imperativa simples.

Pretendemos com este trabalho melhorar o nosso conhecimento no que toca a gramáticas e assembly .

Dentro das tarefas que nos foram propostas **temos presente todas as funcionalidades exceto a inclusão de array's e matrizes pois escolhemos fazer subprogramas.**

Iremos então começar por indicar a sintaxe da nossa linguagem, mostrar a gramática utilizada , o nosso raciocínio por detrás desta e de seguida iremos mostrar as regras de tradução para assembly.

Além disso iremos apresentar alguns exemplos que mostram as capacidades do nosso compilador.

2.1 Estrutura do Relatório

Capítulo 3

Sintaxe do código

Declarações

```
inteiro x em que x é uma variável do tipo inteiro

nice () (define nice ... return x )

declarar a função nice que devolve a variável x
(Sempre que nice for invocado tem de ser através de nice(), além disso é de notar que
que as variaveis que são usadas dentro da função , têm de estar declaradas antes desta
(é possivel declarar variaveis dentro da auxiliar
mas não é recomendável por causa dos problemas que poderá ter
com PUSHG'S E STOREG'S associados com return's e atribuições)
```

Aritmética

```
+ (x,y) soma de x com y (x+y)
- (x,y) subtração de x com y (x-y)
* (x,y) multiplicação de x com y (x*y)
/ (x,y) divisão de x com y (x/y)
% (x,y) resto da divisão de x com y (x%y)
```

Logica

```
>= (x,y) x >= y
> (x,y) x > y
<= (x,y)x <= y
< (x,y) x < y
== (x,y) x == y
~= (x,y) x ~= y
~ (x) negação de x
|| (x,y) x ou y
&& (x,y) x e y
```

Atribuições

```
=
é para atribuições (A variável que receberá a atribuição terá de ser previamente declarada)
x=1
x toma o valor de 1
x=y
x toma o valor de y
inteiro x = 1
declara-se x como inteiro e é lhe atribuído o valor 1
inteiro x = y
declara-se x como inteiro e é lhe atribuído o valor de y (é de uma certa forma inutil porque
na fase de declarações das variaveis têm o valor 0)
```

IF's

```
x representa uma condição
if ( x ) then y end
Se x então fazemos y
if ( x ) then y else z end
Se x então fazemos y senão fazemos z
```

Ciclo while

```
while (x) do y end
enquanto x é verdade fazemos y
```

Leitura do stdin

```
x = <<
ler standard input e guardar em x o resultado
```

Escrita no stdout

```
>> x
escrever o valor de x no standard output
```

Imprimir uma Palavra \ Frase

```
print ( x )
```

Escreve a palavra ou frase x no standard output (Não imprime numeros, apenas palavras e frases).
Nome de varíaveis e as palavras/frases não podem conter letras maiusculas.

Capítulo 4

Gramática usada

A gramática implementada utiliza o método recursivo à esquerda.
Os símbolos terminais utilizados na gramática independente de contexto implementada para reconhecer a linguagem foram: 'NUMERO', 'NOME', 'ATR', 'DEFINE', 'RETURN', 'LER', 'SE', 'ENTAO','FIM', 'SENAO', 'ENQUANTO', 'ESCREVER', 'VIRG', 'MAIOR', 'MENOR', 'MAIORI', 'MENORI', 'IGUAL', 'DIF', 'E', 'OU', 'NEG','+','-','*','/','(',')','PRINT';
(É possível ver a sua implementação no apendice A)

Gramática que traçamos para a resolução deste trabalho tem as seguintes regras de derivação:

```
Programa : Corpo
  | Glob
  | Glob Corpo

  Corpo : Ope
        | Corpo Ope

  Glob: Decl
  | Glob Decl

  Decl : INT NOME
  | INT NOME ATR NUMERO
  | INT NOME ATR NOME
  | Fundecl DEFINE NOME Programa RETURN NOME ')'
  | Fundecl DEFINE NOME Programa RETURN NUMERO ')'
  | Fundecl DEFINE NOME RETURN NUMERO ')'

  Fundecl : NOME '(' ')' '('

 Ope : Atrib
    | Se
    | Enquanto
    | Escrever
    | PRINT '(' Frase ')'
```

```
Frase : NOME
      | Frase NOME

Atrib : NOME ATR Expr
      | NOME ATR LER

Se : SE Cond ENTAO Corpo FIM
   | SE Cond ENTAO Corpo SENAO Corpo FIM

Enquanto : ENQUANTO Cond FAZ Corpo FIM

Escrever : ESCREVER Expr

Expr : NOME `(` `)`
     | Var
     | NUMERO
     | `+` `(` Expr VIRG Expr `)`
     | `-` `(` Expr VIRG Expr `)`
     | `*` `(` Expr VIRG Expr `)`
     | `/` `(` Expr VIRG Expr `)`
     | `%` `(` Expr VIRG Expr `)`
     | Cond
     | `(` Expr `)`

Cond : MAIOR `(` Expr VIRG Expr `)`
     | MENOR `(` Expr VIRG Expr `)`
     | MAIORI `(` Expr VIRG Expr `)`
     | MENORI `(` Expr VIRG Expr `)`
     | IGUAL `(` Expr VIRG Expr `)`
     | DIF `(` Expr VIRG Expr `)`
     | E `(` Cond VIRG Cond `)`
     | OU `(` Cond VIRG Cond `)`
     | NEG `(` Cond `)`
     | '(' Cond ')'

Var : NOME
```

Capítulo 5

Decisões no desenho da gramática

O programa pode ter só a declaração de variaveis globais (Glob), só Corpo ou ambos.

Dentro do das variaveis globais temos as declarações (Decls).

É possível declarar inteiros e funções das diferentes formas
inteiro x
inteiro x = y
nome () (define nome corpo return numero) função nome faz o programa x e devolve um numero
nome () (define nome corpo return y) função nome faz o programa x e devolve a variavel y
nome () (define nome return numero) função nome que apenas devolve um numero
A recursividade premite mais do que uma declaração num programa.
(Fundecl é do tipo nome () (, isto destingue a invocação nome() da função da declaração nome()( )

Dentro do corpo temos diferentes operações (Ope)
Atribuições (Atrib)
If's (Se)
while's (Enquanto)
>> (Escrever)
print (Frase) Frase é uma palavra ou um encadeamento de palavras. A recursividade premite mais do que uma operação num programa.

Começaremos pelas atribuições.
Uma atribuição pode ser do tipo.
x= expressão (explicaremos o que é uma expressão mais á frente (Expr))
x= << (em que << é a leitura do stdin)

De seguida , If's
Podem ser de 2 tipos:
if condições then Corpo end
if condições then Corpo else Corpo end
condições(Cond) aparecerão mais á frente

Os while's
Podem ser apenas de uma forma:
while condições do Corpo end

Por fim o Escrever
É da forma
>> expressão (possibilita a escrita no stdout)

Passamos então para as expressões e as condições, já pudemos ver anteriormente que estas serão bastante utilizadas.

Começaremos pelas expressões (Expr).
Uma expressão pode ser:
nome () ou seja uma chamada/invocação de uma função
x ou seja uma variável
1 ou seja um número
+(expressão,expressão) uma soma entre duas expressões (usamos recursividade de modo a ser possivel encadear outras operações)
-(expressão,expressão) uma subração entre duas expressões
(expressão,expressão) uma multiplicação entre duas expressões
/(expressão,expressão) uma divisão entre duas expressões
%(expressão,expressão) resto de uma divisão entre duas expressões
Uma expressão pode ser uma condição

Seguimos para as Condições (Cond),
As condições encontram-se sempre dentro de parentesis
(x) em que x é uma ou mais condições.

Descemos um nível para o Cond.
Dentro deste temos:
>(expressão,expressão) primeira expressão maior que a segunda
<(expressão,expressão) primeira expressão menos que a segunda
>=(expressão,expressão) primeira expressão maior ou igual que a segunda
<=(expressão,expressão) primeira expressão menor ou igual que a segunda
==(expressão,expressão) primeira expressão igual que a segunda
~=(expressão,expressão) primeira expressão diferente que a segunda
&&(condição,condição) primeira condição e a segunda
||(condição,condição) primeira condição ou a segunda
~ (condição) negação da condição
(condição) sintaxe da condição.
Recursividade premite encadear outra condições.

Para terminar temos Var que é apenas um NOME.

Os restantes NOME NUMERO etc são possíveis de se perceber através do apendice A onde o lex expecifica o que estas representam.

Capítulo 6

Regras de transformação para Assembly

Seguindo o apendice B podemos ver como é o processo de transformação para assembly.

Na declaração de variáveis

inteiro x
fazemos PUSHI 0

inteiro x = numero
fazemos PUSHI numero

inteiro x = y
fazemos PUSHG (posição da stack onde este se encontra y)

Declaração de funções

funcao () (define funcao ... return x)
Retiramos o STOP que vem do programa que está dentro da auxiliar, Damos PUSGH da posição da stack onde se encontra a variavel que queremos que seja retornada e depois fazemos RETURN

funcao () (define funcao ... return numero)
Mesmo processo que a anterior mas em vez de PUSHG damos PUSHI e o numero

funcao () (define funcao return numero)
Mesmo processo que a anterior mas desta vez não é necessário remover o STOP mas é necessário acrescentar no inicio o START

Imprimir uma palavra
print (Frase)
Dá-mos PUSH S "metemos o que se encontra na frase "e depois WRITES

If's

if cond then corpo end
Codigo assembly correspondente á condição depois colocamos JZ l(label deste if) depois vem o codigo do corpo e por fim mete l(label do fim do if): NOP

if cond then corpo else corpo end
Parte do if
Codigo assembly correspondente á condição depois colocamos JZ l(label) depois vem o codigo do corpo , de seguida um JUMP l(label) f

Parte do else
Continuamos com l(label): NOP depois vem o código associado ao corpo do else e por fim este coloca o l(label)f: NOP
Isto faz com que se a condição antes de JZ não verificar ele vai para o else , mas se for verificada ele faz o codigo depois do JZ e por fim salta para depois do codigo do else.

While's

l(label)c: NOP , codigo que equivale á condição, depois JZ l(label)f ,codigo do corpo , depois JUMP para l(label)c, e por fim l(label)f: NOP
Enquanto a condição antes de JZ for verificada ele entra no corpo e no fim salta para o inicio quando esta não se verificar ele salta para o fim.

(A cada if ou while que aparecer a label que usa é incrementada.
Exemplo: 2 if's e 1 while
o primeiro if vai ter label l0 e como final l0f
o segundo if vai ter label l1 e como final l1f
o while vai ter label l2c e como final l2f
Isto é para não causar problemas nos JUMP'S, assim não saltamos para o sitio errado.)

Atribuições:

x = expr
pega no codigo da expressão e depois coloca STOREG e a posição da stack de x

x = <<
Coloca um READ e um ATOI depois UM STOREG e a posição da stack de x

Escrita:

>> expr
Depois do codigo da expressão, ele dá WRITEI , PUSHS "\n"e WRITES

Invocação de uma função:
nome()
Dá-mos PUSHA nome() e depois CALL

Expressões:

+(expr,expr)
Depois do codigo da primeira e da segunda expressão, colocamos ADD.

-(expr,expr)
Depois do codigo da primeira e da segunda expressão, colocamos SUB.

*(expr,expr)
Depois do codigo da primeira e da segunda expressão, colocamos MUL.

/(expr,expr)
Depois do codigo da primeira e da segunda expressão, colocamos DIV.

%(expr,expr)
Depois do codigo da primeira e da segunda expressão, colocamos MOD.

Condições:

>(expr,expr)
Depois do codigo da primeira e da segunda expressão, colocamos SUP.

<(expr,expr)
Depois do codigo da primeira e da segunda expressão, colocamos INF.

>=(expr,expr)
Depois do codigo da primeira e da segunda expressão, colocamos SUPEQ.

<=(expr,expr)
Depois do codigo da primeira e da segunda expressão, colocamos INFEQ.

==(expr,expr)
Depois do codigo da primeira e da segunda expressão, colocamos EQUAL.

~=(expr,expr)
Depois do codigo da primeira e da segunda expressão, colocamos EQUAL e depois NOT.

&&(cond,cond)
Depois do codigo da primeira e da segunda condição, colocamos ADD , PUSHI 2 e EQUAL.
Verdade e Verdade é o mesmo que 1 e 1 ou seja a soma das duas condição tem de dar 2.

||(cond,cond)
Depois do codigo da primeira e da segunda condição, colocamos ADD, PUSHI 1 e SUPEQ.
Se pelo menos um for Verdade a soma das duas condições tem de dar pelo menos 1 ou seja ¿=1.

~(Cond)
Depois do codigo da condição, colocamos NOT.

Por fim temos que sempre que uma variável for invocada nas expressões, temos de dar PUSHG e a posição que essa variável se encontra na stack.

Coisas a notar:
Dentro do código existem 6 pontos importantes.
parser.sucess : Variável do tipo lógico que ajudará a tratar dos erros.
parser.registos : Dicionário que terá como chave o nome das variáveis e como valor, a posição na stack.
parser.labels : Variável do tipo inteiro que irá ser incrementada conforme os if's e while's que aparecerão.
parser.gp : Variável do tipo inteiro que irá ser incrementada quando forem declaradas variáveis.
parser.codigoprincipal : Variável do tipo String que terá todo o texto em assembly.
parser.codigoauxilar : Variável do tipo String que terá todo o texto das funções auxiliares.

Capítulo 7

Testes

Para executar o programa temos de colocar neste formato : Programa ficheiro Input ficheiro Output
Exemplo: yacctrabalho.py input.txt output.txt
Se quisermos que o resultado apareça no stdout do que no ficheiro output temos que colocar no seguinte formato:
Programa ficheiro Input
Exemplo: yacctrabalho.py input.txt

7.1 Primeiro teste:

Código na linguagem que definimos:

```
inteiro a = 1
inteiro b = 3
inteiro c

c = /(+(a,b),2)
>> c
```

Código Assembly resultante:

```
PUSHI 1
PUSHI 3
PUSHI 0
START
PUSHG 0
PUSHG 1
ADD
PUSHI 2
DIV
STOREG 2
PUSHG 2
WRITEI
PUSHS "\n"
WRITES
STOP
```

7.2 Segundo teste:

Código na linguagem que definimos:

```
inteiro x
inteiro y

ok () (define ok
       return 1)

x = <<

if (==(x,ok()))
then >> 1
else >> 0
end
```

Código Assembly resultante:

```
PUSHI 0
PUSHI 0
START
READ
ATOI
STOREG 0
PUSHG 0
PUSHA ok
CALL
EQUAL
JZ l0
PUSHI 1
WRITEI
PUSHS "\n"
WRITES
JUMP l0f
l0: NOP
PUSHI 0
WRITEI
PUSHS "\n"
WRITES
l0f: NOP
STOP

ok:
START
PUSHI 1
RETURN
```

7.3 Terceiro teste:

Código na linguagem que definimos:

```
inteiro x
inteiro y

x = <<
y = 8008135
print ( indique a palavra passe )

while (~=(x,y)) do
x = <<
end

print( palavra passe correta inserida )
```

Código Assembly resultante:

```
PUSHI 0
PUSHI 0
START
READ
ATOI
STOREG 0
PUSHI 8008135
STOREG 1
PUSHS "indique a palavra passe \n"
WRITES
l0c: NOP
PUSHG 0
PUSHG 1
EQUAL
NOT
JZ l0f
READ
ATOI
STOREG 0
JUMP l0c
l0f: NOP
PUSHS "palavra passe correta inserida"
WRITES
STOP
```

7.4 Quarto teste:

Código na linguagem que definimos:

```
inteiro n
inteiro x
zero() ( define zero
         x = 0
         return x )

n = <<
while (>=(n,zero())) do
    >> n
    n = -(n,1)
end
```

Código Assembly resultante:

```
PUSHI 0
PUSHI 0
START
READ
ATOI
STOREG 0
l0c: NOP
PUSHG 0
PUSHA zero
CALL
SUPEQ
JZ l0f
PUSHG 0
WRITEI
PUSHS "\n"
WRITES
PUSHG 0
PUSHI 1
SUB
STOREG 0
JUMP l0c
l0f: NOP
STOP

zero:
START
PUSHI 0
STOREG 1
PUSHG 1
RETURN
```

7.5 Quinto teste:

Código na linguagem que definimos:

```
inteiro a
inteiro b
inteiro c

fact12() ( define fact12
          a = 12
          b = 1
          while (>(a,1)) do
              b=*(b,a)
              a=-(a,1)
          end
          return b )

fact13() ( define fact13
           a = fact12()
           a = *(a,13)
           return a )


a = fact13()
>>a
```

Código Assembly resultante:

```
PUSHI 0
PUSHI 0
PUSHI 0
START
PUSHA fact13
CALL
STOREG 0
PUSHG 0
WRITEI
PUSHS "\n"
WRITES
STOP

fact12:
START
PUSHI 12
STOREG 0
PUSHI 1
STOREG 1
l0c: NOP
```

```
PUSHG 0
PUSHI 1
SUP
JZ l0f
PUSHG 1
PUSHG 0
MUL
STOREG 1
PUSHG 0
PUSHI 1
SUB
STOREG 0
JUMP l0c
l0f: NOP
PUSHG 1
RETURN

fact13:
START
PUSHA fact12
CALL
STOREG 0
PUSHG 0
PUSHI 13
MUL
STOREG 0
PUSHG 0
RETURN
```

Capítulo 8

Conclusão

Este trabalho apesar de ser mais desafiante que o anterior, foi muito mais interessante. Tivemos algumas dificuldades, principalmente no yacc, o compilador não é uma ferramente muito fiável para encontrar erros, houve por exemplo um caso em que nos tinhamos enganado no dicionário que pretendiamos usar e perdemos algum tempo porque o erro que este indicava era completamente diferente e fez-nos desviar o foco para um outro caminho que não era o pretendido. Código Assembly era algo que tinhamos trabalhado muito pouco no nosso percurso académico (excluindo esta cadeira), nunca nos foi pedido um domínio geral deste , por isso tivemos de relembrar certas noções, o site `https://ewvm.epl.di.uminho.pt/` ajudou mesmo muito neste aspeto, sem ele ia ser muito díficil a realização deste trabalho

No que toca ao latex, neste trabalho já tinhamos um maior conforto , não tivemos de investir tanto tempo para aprender o seu uso. É uma ferramente útil e o facto de nos ter sido requesitado o seu uso na realização deste relatório, ajudou a familiarizar mais.

Colocamos em prova conhecimentos de cadeiras de outros anos (Automatos) que na altura pensamos ser um pouco abstrato e graças a este trabalho/Cadeira vimos o quão importante as gramáticas são , ajudou sem dúvida a expandir o nosso conhecimento.

Conseguimos realizar o pretendido, um programa capaz de declarar variáveis, efetuar operações aritméticas, relacionais e lógicas. Temos também a possibilidade de atribuições e a leitura e escrita no standard input e output, respetivamente.

Por fim temos a capacidade de realizar ciclos while , if's e de declarar e invocar subprogramas que têm duas formas possíveis de retorno, uma variavél do tipo inteiro ou um numero inteiro.

Apêndice A

Código do Analisador léxico/Lex

Este é o codigo associado ao analisador léxico/Lex do programa.

```
import ply.lex as lex
import sys

tokens = (
    'NOME',
    'INT',
    'MAIOR',
    'MENOR',
    'IGUAL',
    'MAIORI',
    'MENORI',
    'NIGUAL',
    'NEG',
    'E',
    'OU',
    'ATR',
    'LER',
    'ESCREVER',
    'SE',
    'ENTAO',
    'SENAO',
    'ENQUANTO',
    'FAZ',
    'FIM',
    'VIRG',
    'DEFINE',
    'RETURN',
    'NUMERO',
    'PRINT'
)

t_VIRG = r','
literals = ['(', ')','+','-','*','/','%']

def t_DEFINE(t):
    r'(?i:define)'
    return t

def t_RETURN(t):
    r'(?i:return)'
    return t
```

```
42
43 def t_PRINT(t):
44     r'(?i:print)'
45     return t
46
47 def t_NUMERO(t):
48     r'\d+'
49     return t
50
51 def t_INT(t):
52     r'(?i:inteiro)'
53     return t
54
55 def t_LER (t):
56     r'<<'
57     return t
58
59 def t_ESCREVER (t):
60     r'>>'
61     return t
62
63 def t_MAIORI(t):
64     r'>='
65     return t
66
67 def t_MAIOR(t):
68     r'\>'
69     return t
70
71 def t_MENORI(t):
72     r'\<='
73     return t
74
75 def t_MENOR(t):
76     r'\<'
77     return t
78
79 def t_IGUAL(t):
80     r'=='
81     return t
82
83 def t_NIGUAL(t):
84     r'~='
85     return t
86
87 def t_NEG (t):
88     r'~'
89     return t
90
91 def t_OU (t):
92     r'\|\|'
93     return t
94
95 def t_ATR (t):
96     r'='
97     return t
98
99 def t_ENTAO(t):
100    r'(?i:then)'
```

```
    return t

def t_SENAO(t):
    r'(?i:else)'
    return t

def t_SE (t):
    r'(?i:if)'
    return t

def t_ENQUANTO(t):
    r'(?i:while)'
    return t

def t_FAZ(t):
    r'(?i:do)'
    return t

def t_FIM(t):
    r'(?i:end)'
    return t

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

def t_NOME(t):
    r'[a-z]\w*'
    return t

t_ignore = ' \r\t\n'

def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)


lexer = lex.lex()
```

Apêndice B

Código do Yacc/Compilador

Este é o codigo associado ao Yacc/Compilador do programa.

```
import ply.yacc as yacc
import sys

from lextrabalho import tokens

def p_Programa_Corpo(p):
    "Programa : Corpo"
    parser.codigoprincipal = f'START\n{p[1]}STOP\n'
    p.parser.codigoprincipal=p.parser.codigoprincipal.replace("None", "")

def p_Programa_Glob(p):
    "Programa : Glob"
    parser.codigoprincipal = f'{p[1]}START\nSTOP\n'
    p.parser.codigoprincipal=p.parser.codigoprincipal.replace("None", "")

def p_Programa_Glob_Corpo(p):
    "Programa : Glob Corpo"
    parser.codigoprincipal = f'{p[1]}START\n{p[2]}STOP\n'
    p.parser.codigoprincipal = p.parser.codigoprincipal.replace("None", "")

def p_Glob(p):
    "Glob    : Decl"
    p[0] = f'{p[1]}'

def p_Glob_Recursiva(p):
    "Glob    : Glob Decl"
    p[0] = f'{p[1]}{p[2]}'

def p_Decl_Int(p): # inteiro x
    "Decl     : INT NOME"
    if p[2] not in p.parser.registos:
        p.parser.registos.update({p[2] : p.parser.gp})
        p[0] = f'PUSHI 0\n'
        p.parser.gp += 1
    else:
        print("Erro: Vari vel  j   existe.")
        parser.sucess = False

def p_Decl_Int_Atr(p): # inteiro x = 2
    "Decl     : INT NOME ATR NUMERO"
    if p[2] not in p.parser.registos:
```

```
        p.parser.registos.update({p[2] : p.parser.gp})
        p[0] = f'PUSHI {p[4]}\n'
        p.parser.gp += 1
    else:
        print("Erro: Vari vel  j   existe.")
        parser.sucess = False

def p_Decl_Nome_Atr(p): #inteiro x = y
    "Decl : INT NOME ATR NOME"
    if p[2] not in p.parser.registos:
     if p[4] in p.parser.registos:
            p.parser.registos.update({p[2]: p.parser.gp})
            p[0] = f'PUSHG {p.parser.registos.get(p[4])}\n'
            p.parser.gp += 1
     else:
        print(f"Erro: A Vari vel {p[4]}  n o est  definida.")
        parser.sucess = False
    else:
        print(f"Erro: A Vari vel {p[2]} j   existe.")
        parser.sucess = False

def p_Decl_Func_VAR(p): # funcao () ( define funcao ... return x )
    "Decl : Fundecl DEFINE NOME Programa RETURN NOME ')'"
    if p[3] not in p.parser.codigoauxiliar:
        if p[6] in p.parser.registos:
          codigov = p.parser.codigoprincipal.replace("STOP\n", "")
          codigov = codigov + f'PUSHG {p.parser.registos.get(p[6])}\n'+"RETURN\n"
          p.parser.codigoauxiliar.update({p[3] :codigov})
          p.parser.registos.update({p[1] : p.parser.gp})
        else:
          print(f"Erro: A Vari vel {p[6]} n o est  definida.")
          parser.sucess = False
    else:
        print("Erro: A fun   o j   existe.")
        parser.sucess = False

def p_Decl_Func_Int(p): # funcao () ( define funcao ... return 2 )
    "Decl : Fundecl DEFINE NOME Programa RETURN NUMERO ')'"
    if p[3] not in p.parser.codigoauxiliar:
        codigoi = p.parser.codigoprincipal.replace("STOP\n", "")
        codigoi =codigoi+f'PUSHI {p[6]}\n'+"RETURN\n"
        p.parser.codigoauxiliar.update({p[3] :codigoi})
        p.parser.registos.update({p[1] : p.parser.gp})
    else:
        print("Erro: A fun   o j   existe.")
        parser.sucess = False

def p_Decl_Func_Int_vazia(p): # funcao () ( define funcao return 2 )
    "Decl : Fundecl DEFINE NOME RETURN NUMERO ')' "
    if p[3] not in p.parser.codigoauxiliar:
        codigoi =p.parser.codigoprincipal+f'START\nPUSHI {p[5]}\n'+"RETURN\n"
        p.parser.codigoauxiliar.update({p[3] :codigoi})
        p.parser.registos.update({p[1] : p.parser.gp})
    else:
        print("Erro: A fun   o j   existe.")
        parser.sucess = False

def p_Fundecl(p):
    "Fundecl : NOME '(' ')' '('"
```

```
101     p[0]=f'{p[1]}{p[2]}{p[3]}'
102
103 def p_Corpo(p):
104     "Corpo     : Ope"
105     p[0] = p[1]
106
107 def p_Corpo_Recursiva(p):
108     "Corpo     : Corpo Ope"
109     p[0] = f'{p[1]}{p[2]}'
110
111 def p_Ope_Atrib(p):
112     "Ope       : Atrib"
113     p[0] = p[1]
114
115 def p_Ope_Escrever(p):
116     "Ope       : Escrever"
117     p[0] = p[1]
118
119 def p_Ope_Se(p):
120     "Ope       : Se"
121     p[0] = p[1]
122
123 def p_Ope_Enquanto(p):
124     "Ope       : Enquanto"
125     p[0] = p[1]
126
127 def p_Ope_PRINT(p):
128     "Ope : PRINT '(' Frase ')' "
129     p[0] = f'PUSHS "{p[3]}"\nWRITES\n'
130
131 def p_Frase_nome(p):
132     "Frase : NOME"
133     p[0] = p[1]
134
135 def p_Frase(p):
136     "Frase : Frase NOME"
137     p[0] = f'{p[1]} {p[2]}'
138
139 def p_Se(p):
140     "Se        : SE Cond ENTAO Corpo FIM"
141     p[0] = f'{p[2]}JZ l{p.parser.labels}\n{p[4]}l{p.parser.labels}: NOP\n'
142     p.parser.labels += 1
143
144 def p_Se_Senao(p):
145     "Se        : SE Cond ENTAO Corpo SENAO Corpo FIM"
146     p[0] = f'{p[2]}JZ l{p.parser.labels}\n{p[4]}JUMP l{p.parser.labels}f\nl{p.parser.
                                                labels}: NOP\n{p[6]}l{p.parser.labels}f:
                                                NOP\n'
147     p.parser.labels += 1
148
149 def p_Enquanto(p):
150     "Enquanto : ENQUANTO Cond FAZ Corpo FIM"
151     p[0] = f'l{p.parser.labels}c: NOP\n{p[2]}JZ l{p.parser.labels}f\n{p[4]}JUMP l{p.parser
                                                .labels}c\nl{p.parser.labels}f: NOP\n'
152     p.parser.labels += 1
153
154 def p_Atrib_expr_Int(p):
155     "Atrib     : NOME ATR Expr"
156     if p[1] in p.parser.registos:
```

```
157             p[0] = f'{p[3]}STOREG {p.parser.registos.get(p[1])}\n'
158     else:
159         print("Erro: Vari vel  n  o  definida.")
160         parser.sucess = False
161 
162 def p_Atrib_Ler(p):
163     "Atrib     : NOME ATR LER"
164     if p[1] in p.parser.registos:
165         p[0] = f'READ\nATOI\nSTOREG {p.parser.registos.get(p[1])}\n'
166     else:
167         print("Erro: Vari vel   n  o  definida.")
168         parser.sucess = False
169 
170 def p_Escrever(p):
171     "Escrever : ESCREVER Expr"
172     p[0] = f'{p[2]}WRITEI\nPUSHS "\\n"\nWRITES\n'
173 
174 def p_Expr_P(p):
175     "Expr      : '(' Expr ')'"
176     p[0] = p[2]
177 
178 def p_Expr_Func(p):
179     "Expr : NOME '(' ')'"
180     nomefunc=p[1]+p[2]+p[3]
181     if nomefunc in p.parser.registos:
182         p[0] = f'PUSHA {p[1]}\nCALL\n'
183     else:
184         print("Erro: Fun   o  n  o  est   definida.")
185         parser.sucess = False
186 
187 def p_Expr_Var(p):
188     "Expr      : Var"
189     p[0] = p[1]
190 def p_Expr_NUMERO(p):
191     "Expr      : NUMERO"
192     p[0] = f'PUSHI {p[1]}\n'
193 
194 def p_Expr_soma(p):
195     "Expr      : '+' '(' Expr VIRG Expr ')' "
196     p[0] = f'{p[3]}{p[5]}ADD\n'
197 
198 def p_Expr_Sub(p):
199     "Expr      : '-' '(' Expr VIRG Expr ')'"
200     p[0] = f'{p[3]}{p[5]}SUB\n'
201 
202 def p_Expr_Mult(p):
203     "Expr      : '*' '(' Expr VIRG Expr ')'"
204     p[0] = f'{p[3]}{p[5]}MUL\n'
205 
206 def p_Expr_Div(p):
207     "Expr      : '/' '(' Expr VIRG Expr ')'"
208     p[0] = f'{p[3]}{p[5]}DIV\n'
209 
210 def p_Expr_Mod(p):
211     "Expr      : '%' '(' Expr VIRG Expr ')'"
212     p[0] = f'{p[3]}{p[5]}MOD\n'
213 
214 def p_Expr_Cond(p):
215     "Expr      : Cond"
```

```
216     p[0] = p[1]
217
218 def p_Cond_P(p):
219     "Cond      : '(' Cond ')'"
220     p[0] = p[2]
221
222 def p_Cond_Maior(p):
223     "Cond      : MAIOR '(' Expr VIRG Expr ')'"
224     p[0] = f'{p[3]}{p[5]}SUP\n'
225
226 def p_Cond_Menor(p):
227     "Cond      : MENOR '(' Expr VIRG Expr ')'"
228     p[0] = f'{p[3]}{p[5]}INF\n'
229
230 def p_Cond_Maiori(p):
231     "Cond      : MAIORI '(' Expr VIRG Expr ')'"
232     p[0] = f'{p[3]}{p[5]}SUPEQ\n'
233
234 def p_Cond_Menori(p):
235     "Cond      : MENORI '(' Expr VIRG Expr ')'"
236     p[0] = f'{p[3]}{p[5]}INFEQ\n'
237
238 def p_Cond_Igual(p):
239     "Cond      : IGUAL '(' Expr VIRG Expr ')'"
240     p[0] = f'{p[3]}{p[5]}EQUAL\n'
241
242 def p_Cond_Nigual(p):
243     "Cond      : NIGUAL '(' Expr VIRG Expr ')'"
244     p[0] = f'{p[3]}{p[5]}EQUAL\nNOT\n'
245
246 def p_Cond_E(p):
247     "Cond      : E '(' Cond VIRG Cond ')'"
248     p[0] = f'{p[3]}{p[5]}ADD\nPUSHI 2\nEQUAL\n'
249
250 def p_Cond_Ou(p):
251     "Cond      : OU '(' Cond VIRG Cond ')'"
252     p[0] = f'{p[3]}{p[5]}ADD\nPUSHI 1\nSUPEQ\n'
253
254 def p_Cond_Neg(p):
255     "Cond      : NEG '(' Cond ')'"
256     p[0] = f'{p[3]}NOT\n'
257
258 def p_Var_Int(p):
259     "Var       : NOME"
260     if p[1] in p.parser.registos:
261             p[0] = f'PUSHG {p.parser.registos.get(p[1])}\n'
262     else:
263         print("Erro: Vari vel  n  o  definida.")
264         parser.sucess = False
265
266 #Erro
267 def p_error(p):
268     print('Syntax error: ', p)
269     parser.sucess = False
270
271
272
273 #Inicio do Parser
274 parser = yacc.yacc()
```

```
275
276 parser.sucess = True
277 parser.registos = {}
278 parser.labels = 0
279 parser.gp = 0
280 parser.codigoprincipal = ""
281 parser.codigoauxiliar ={}
282
283
284
285 if len(sys.argv) > 1:
286     with open(sys.argv[1],'r') as file:
287         input = file.read()
288         parser.parse(input)
289         if parser.sucess:
290             if len(sys.argv) > 2:
291                 with open(sys.argv[2], 'w') as output:
292                     for t in parser.codigoauxiliar:
293                         parser.codigoprincipal=parser.codigoprincipal+"\n"+t+":\n"+parser.
                                                                                codigoauxiliar.get(t)
294                     output.write(parser.codigoprincipal)
295                     print(f"\n0 ficheiro {sys.argv[1]} foi compilado com sucesso.\n\n0
                                                                    output ficou guardado em
                                                                    {sys.argv[2]}.\n")
296             else:
297                 for t in parser.codigoauxiliar:
298                     parser.codigoprincipal = parser.codigoprincipal + t + ":\n" + parser.
                                                                    codigoauxiliar.get(t)
299                 print(parser.codigoprincipal)
300         else:
301             print("\nErro ao compilar.\n")
```