Processamento de Linguagens e Compiladores (3º ano de Curso)

Trabalho Prático 2

Relatório de Desenvolvimento

David Alberto Agra
(a95726)

João Luís da Cruz Pereira
(a95375)

João Manuel Franqueira da Silva
(a91638)

10 de janeiro de 2024

Resumo

Este relatório trata-se do segundo projeto da unidade curricular de *Processamento de Linguagens e Compiladores*. Contém uma linguagem de programação criada pelos alunos, o seu compilador e todo o desenvolvimento por detrás do mesmo. É coberta também a geração de código assembly que pode ser corrido numa máquina virtual. Utilizou-se Lex e Yacc em Python para resolver o projeto.

Conteúdo

Capítulo 1

Introdução

No âmbito da disciplina de Processamento de Linguagens e Compiladores, foi-nos proposto o desenvolvimento de uma linguagem de programação simples, que nos permita efetuar instruções básicas, e de um compilador para reconhecer programas escritos nessa linguagem, gerando código assembly que preserva a semântica do programa fonte e que possa ser executado numa máquina virtual. (`https://ewvm.epl.di.uminho.pt/`).

1.1 Objetivo

Este relatório tem como principal objetivo informar o leitor sobre como se poderia escrever um programa fonte para a nossa linguagem de programação e também, com ainda mais importância, como é que o nosso compilador foi feito.

1.2 Estrutura do Relatório

O relatório está organizado por diferentes fases. Começamos por analisar o problema que nos foi dado no capítulo 2, de seguida expomos de uma forma visual e simples como é que um programa para a nossa linguagem poderia ser escrito no capítulo 3. Explicamos de forma relativamente sucinta como é que desenvolvemos cada um dos objetivos propostos no capítulo 4. No capítulo 5 mostramos alguns exemplos de programas e o seu respetivo código assembly. Temos também alguns exemplos de mensagens de erro caso o ficheiro de entrada não esteja correto. Por fim, no capítulo 6, terminamos o relatório com uma pequena conclusão e o trabalho futuro.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

Pretende-se definir uma linguagem de programação simples e imperativa, capaz de:

- Declarar variáveis atómicas do tipo inteiro.
- Efetuar instruções algorítmicas básicas.
- 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.
- 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.

  Deve ser desenvolvido um compilador para a linguagem criada com base na **GIC** criada e com recurso aos módulos Yacc/Lex do PLY/Python. O compilador deve gerar **pseudo-código**, Assembly da Máquina Virtual VM (`https://ewvm.epl.di.uminho.pt/`).

Capítulo 3

Concepção/Desenho da Resolução

Na fase inicial do design da nossa linguagem decidimos que queríamos algo familiar e fácil de escrever por isso reduzimos as nossas opões para algo baseado em Python ou C, primeiro experimentamos com Python mas houve algumas características de C que preferíamos como por exemplo o uso de chavetas e de pontos e virgula para delimitar as declarações, as declarações de tipo das variáveis, entre outros. O resultado final foi uma linguagem parecida com C mas com alguma inspiração de Python.

3.1 Estruturas de Dados

3.1.1 Declarações e atribuições

Comando	**Função**
int n	declarar n como inteiro
int mat[10][10]	declarar mat como uma matriz 10x10
int arr[10]	declarar arr como um array de tamanho 10
n=5	atribuir o valor 5 a n
n=soma(a, b)	atribuir valor que é retornado pela função *soma* a n

3.1.2 Operações

Aritméticas

Comando	**Função**
`x + y`	somar x com y
`x - y`	subtrair y a x
`x * y`	multiplicar x por y
`x / y`	dividir y a x (divisão inteira)

Relacionais

Comando	Função
`x > y`	calcula x maior do que y
`x >= y`	calcula x maior ou igual do que y
`x < y`	calcula x menor do que y
`x <= y`	calcula x menor ou igual do que y
`x == y`	calcula x igual a y
`x != y`	calcula x diferente de y

Lógicas

Comando	Função
`x && y`	calcula o resultado de x e y
`x || y`	calcula o resultado de x ou y

3.1.3 Instruções condicionais

If-then

Executa S se x for verdadeiro.

```
1    if (x) {S}
```

If-then-Else

Executa S1 se x for verdadeiro, caso contrário, executa S2

```
1    if (x) {S1}
2    else {S2}
```

Em ambos os casos, se S, S1, S2 forem compostos apenas por uma única declaração, temos a opção de escrever da seguinte forma:

```
1    if (x) S
2
3    if (y) S1
4    else S2
```

3.1.4 Instruções cíclicas

While-do

Enquanto x for verdadeiro, executa S.

```
1    while (x) {S}
```

3.1.5 Indexação

Para aceder à posição 2 do array arr.

```
1     arr[2]
```

3.1.6 Input-Output

Comando	Função
`printi(x)`	escreve o resultado de x no output
`prints("hello")`	escreve a string "hello"no output
`println()`	escreve uma linha no output
`input()`	lê do input

3.1.7 Programas

Função com parâmetros.

```
1 def function1( x ) int {
2     return x + 1;
3 }
```

Função sem parâmetros .

```
1 def function2( ) void {
2     prints("isto-e-uma-funcao");
3     return ;
4 }
```

3.2 Desenho da gramática

A gramática implementada é constituída pelas seguintes regras de derivação:

```
1    S               : program
2
3    program         : declist funlist codeblock
4
5    codeblock       : stmlist
6                    |
7
8    declist         : dec declist
9                    |
10
11   dec             : dec_int
12                   | dec_arr
13                   | dec_mat
14
15   dec_int         : INT ID SC
16
17   dec_arr         : INT ID LBRACKET NUM RBRACKET SC
18
19   dec_mat         : INT ID LBRACKET NUM RBRACKET LBRACKET NUM RBRACKET SC
20
21   funlist         : fun funlist
22                   |
23   fun             : DEF ID LPAREN idlist RPAREN INT LCURLY stmlist RETURN exprl SC
        RCURLY
24                   | DEF ID LPAREN idlist RPAREN VOID LCURLY stmlist RETURN SC RCURLY
25
26   idlist          : ID cont
27                   |
28   cont            : COMMA ID cont
29                   |
30
31   stmlist         : stmt stmlist
32                   | stmt
33
34   stmt            | PRINTI LPAREN exprl RPAREN SC
35                   | PRINTLN LPAREN RPAREN SC
36                   | PRINTS LPAREN STRING RPAREN SC
37                   | WHILE LPAREN exprl RPAREN block
38                   | ID ASSIGN exprl SC
39                   | ID LBRACKET exprl RBRACKET ASSIGN exprl SC
40                   | ID LBRACKET exprl RBRACKET LBRACKET exprl RBRACKET ASSIGN exprl
                       SC
41                   | IF LPAREN exprl RPAREN block ELSE block
42                   | IF LPAREN exprl RPAREN block
43                   | INPUT LPAREN RPAREN SC
44                   | exprl SC
45
46   block           : LCURLY stmlist RCURLY
47                   | LCURLY stmt RCURLY
48                   | stmt
49
```

```
50      exprl           : expr oprl exprl
51                      | expr
52
53      expr            : expr opra term
54                      | term
55
56      term            : term oprm factor
57                      | factor
58
59      factor          : LPAREN expr RPAREN
60                      | ID
61                      | NUM
62                      | NOT exprl
63                      | NEG exprl
64                      | ID LBRACKET exprl RBRACKET
65                      | ID LBRACKET exprl RBRACKET LBRACKET exprl RBRACKET
66                      | ATOI LPAREN argatoi RPAREN
67                      | ID LPAREN exprllist RPAREN
68                      | TRUE
69                      | FALSE
70
71      exprllist       : exprl contexprllist
72                      |
73
74
75      contexprllist   : COMMA exprl contexprllist
76                      |
77
78      argatoi         : STRING
79                      | INPUT LPAREN RPAREN
80
81      opra            : ADD
82                      | SUBT
83                      | MULT
84                      | DIV
85
86      oprl            : EQ
87                      | GEQ
88                      | LEQ
89                      | LT
90                      | GT
91                      | NEQ
92                      | AND
93                      | OR
```

Capítulo 4

Desenvolvimento

4.1 Declarações e Atribuições

```
def p_dec_int(p):
        'dec_int : INT ID SC'
        if parser.success:
                name = p[2]
                if name in parser.dict['funcs']:
                        print(f'Error: Identifier already declared as function {name}.')
                        parser.success = False
                elif name in parser.dict['vars']:
                        print(f'Error: Identifier already declared as variable {name}')
                        parser.success = False
        if parser.success:
                parser.dict['vars'].update({name:
                        {'size': 0,
                         'count': parser.count
                        }
                 }
                )
                parser.count += 1
        p[0] = 'PUSHI 0\n'
```

Para declarar um inteiro, verificamos se o seu **ID (p[2])** já foi previamente declarado, como variável ou como função. Caso tal não se verifique, adicionamos o seu **ID** ao dicionário das variáveis, associando-lhe a sua posição na stack **(parser.count)**. Por último incrementamos o **parser.count** com o número de células da stack necessárias para guardar um inteiro, neste caso, uma.

```
def p_def_arr(p):
        'dec_arr : INT ID LBRACKET NUM RBRACKET SC'
        if parser.success:
                name = p[2]
                size = p[4]
                if name in parser.dict['funcs']:
                        print(f'Error: Identifier already declared as function {name}.')
```

```
27                          parser.success = False
28                  elif name in parser.dict['vars']:
29                          print(f'Error: Identifier already declared as variable {name
                                }')
30                          parser.success = False
31          if parser.success:
32                  parser.dict['vars'].update({name:
33                          {'size': size,
34                           'count': parser.count
35                          }
36                  }
37                  )
38                  parser.count += size
39          p[0] = f'PUSHN {size}\n'
```

Na declaração de um array, primeiro verificamos se o nome atribuído ao array **(p[1])** já foi previamente declarado, como função ou como variável, se tal não se verificar, guardamos o seu **ID** no dicionário das variáveis, associando-lhe o seu tamanho **(p[4])** e a posição da stack em que se encontra **(parser.count)**. Por ultimo, incrementamos o valor de **parser.count** com o tamanho do array **(p[4])**.

```
40 def p_def_mat(p):
41          'dec_mat : INT ID LBRACKET NUM RBRACKET LBRACKET NUM RBRACKET SC'
42          if parser.success:
43                  name = p[2]
44                  row = p[4]
45                  col = p[7]
46                  if name in parser.dict['funcs']:
47                          print(f'Error: Identifier already declared as function {name
                                }.')
48                          parser.success = False
49                  elif name in parser.dict['vars']:
50                          print(f'Error: Identifier already declared as variable {name
                                }')
51                          parser.success = False
52          if parser.success:
53                  parser.dict['vars'].update({name:
54                          {'size': row*col,
55                           'count': parser.count,
56                           'row':row,
57                           'col':col
58                          }
59                  }
60                  )
61                  parser.count += row*col
62          p[0] = f'PUSHN {row*col}\n'
```

No caso das matrizes, como anteriormente, verificamos se alguma função ou variável já foi declarada com o **ID** da matriz, se não, adicionamos o seu **ID** ao dicionário das variáveis, associando-lhe a sua posição na stack **(parser.count)**. Incrementamos o **parser.count** pelo número de células na stack necessárias para guardar a matriz, neste caso, o número de linhas multiplicado pelo número de colunas **(p[4] * p[7])**.

```
63 def p_stmt5(p):
64         'stmt : ID ASSIGN exprl SC'
65         if parser.success:
66                 name = p[1]
67                 if name in parser.dict['vars']:
68                         address = parser.dict['vars'][name]['count']
69                 elif name in parser.dict['funcs']:
70                         print(f'Error: Identifier is function not variable {name}')
71                         parser.success = False
72                 else:
73                         print('Error: Variable not declared.')
74                         parser.success = False
75         if parser.success:
76                 p[0] = f'{p[3]}STOREG {address}\n'
```

Para atribuir algum valor a alguma variável, primeiramente precisamos de verificar se o nome da variável já foi previamente declarado, se sim, então vamos obter a posição da stack dessa variável guardada no dicionário, na variável address, calculamos o valor da atribuição **(p[3])**, e produzimos o código para a máquina virtual guardar o valor em **gp[address] (STOREG address)**. Como a posição na stack associado a este **ID** é única, garantimos que conseguimos sempre manusear a variável correta.

4.2 Operações

Exemplo de uma soma entre uma variável e um inteiro:

```
77 def p_exprl2(p):
78         'exprl : expr'
79         if parser.success:
80                 p[0] = p[1]
81
82 def p_expr1(p):
83         'expr : expr opra term'
84         if parser.success:
85                 p[0] = p[1] + p[3] + p[2]
86
87 def p_opra1(p):
88         'opra : ADD'
89         if parser.success:
90                 p[0] = 'ADD\n'
91
92 def p_expr2(p):
93         'expr : term'
94         if parser.success:
95                 p[0] = p[1]
96
97 def p_term2(p):
98         'term : factor'
99         if parser.success:
100                p[0] = p[1]
101
102 def p_factor2(p):
103         'factor : ID'
```

```
104         if parser.success:
105                 name = p[1]
106                 if name in parser.dict['vars']:
107                         address = parser.dict['vars'][name]['count']
108                 elif name in parser.dict['funcs']:
109                         print(f'Error: Identifier is function not variable {name}')
110                         parser.success = False
111                 else:
112                         print('Error: Variable not declared.')
113                         parser.success = False
114         if parser.success:
115                 p[0] = f'PUSHG {address}\n'
116
117 def p_factor3(p):
118         'factor : NUM'
119         if parser.success:
120                 p[0] = f'PUSHI {p[1]}\n'
```

Vimos que podemos reduzir <**exprl2**> a <**term**> <**opra**> <**term**> e, pela regra <**p_expr1**> temos que **p[0]=p[1]+p[3]+p[2]**, ou seja, primeiro produzimos o código para a variável associada ao **ID**, depois para o número, e finalmente para a operação em causa.

Para calcular o valor da variável, necessitamos de verificar se a variável já foi previamente declarada, se sim, então obtemos a sua posição na stack associada ao seu **ID** em address, e obtemos o seu valor, guardado na máquina virtual em **gp[address]**, através de **'PUSHG address'**.

Quando nos deparamos com um inteiro, apenas precisamos de fazer **push** para a stack de ele próprio **(PUSHI p[1])**. No fim temos a string:

```
STOREG address
PUSHI n
ADD
```

Exatamente como pretendíamos, para calcular o valor de tal expressão na máquina virtual. As outras operações seguem todas um processo semelhante a este.

4.3 Instruções Condicionais

```
1 def p_stmt9(p):
2         'stmt : IF LPAREN exprl RPAREN block'
3         if parser.success:
4                 lbl_end = parser.label
5                 parser.label += 1
6                 p[0] = f'{p[3]}JZ lbl{lbl_end}\n{p[5]}lbl{lbl_end}:\n'
```

Depois de analisada a expressão lógica **(p[3])** e caso esta tenha um valor igual a 0, é efetuado um salto para a **label** colocada no fim da string, caso contrário, o salto não é realizado e o corpo do **if (p[5])** é executado. O valor de **parser.label** é sempre incrementado, o que é essencial para fazermos o salto para a parte correta do código.

```
5 def p_stmt8(p):
6         'stmt : IF LPAREN exprl RPAREN block ELSE block'
```

```
7         if parser.success:
8                 lbl_else = parser.label
9                 lbl_end = parser.label + 1
10                parser.label += 2
11                p[0] = f'{p[3]}JZ lbl{lbl_else}\n{p[5]}JUMP lbl{lbl_end}\nlbl{
                      lbl_else}:\n{p[7]}lbl{lbl_end}:\n'
```

Começamos também por analisar a expressão lógica **(p[3])** e caso esta tenha um valor igual a 0, é efetuado um salto para a **label** colocada no início do código **else**, pois se a condição é falsa, executa-se o **else**. Se o valor for diferente de 0, então executamos o código do **if**, e no fim, efetuamos um salto para a segunda **label** colocada no fim do código, pois não queremos executar o código do **else**. Como foram usadas duas **labels** o valor de **parser.label** é incrementado em duas unidades.

4.4 Instruções Cíclicas

```
121 def p_stmt4(p):
122         'stmt : WHILE LPAREN exprl RPAREN block'
123         if parser.success:
124                 lbl_ini = parser.label
125                 lbl_end = parser.label + 1
126                 parser.label += 2
127                 p[0] = f'lbl{lbl_ini}:\n{p[3]}JZ lbl{lbl_end}\n{p[5]}JUMP lbl{lbl_ini
                        }\nlbl{lbl_end}:\n'
```

Para efetuar o **while-do**, usaremos duas **labels**, uma antes da expressão lógica **(p[3])** ser analisada e outra depois do corpo do **while**. Ao analisarmos a expressão lógica, se esta tiver valor igual a 0, efetuamos um salto para a segunda **label**, nunca entrando no ciclo. Caso a expressão for verdadeira, então efetuamos o corpo do ciclo **(p[5])**, e quando terminado, efetuamos um salto para voltar à condição do ciclo. Este processo repete-se até a condição ser falsa. Criamos duas **labels**, logo, **parser.label** é incrementado em 2 unidades.

4.5 Indexação

```
128 def p_factor6(p):
129         'factor : ID LBRACKET exprl RBRACKET'
130         if parser.success:
131                 name = p[1]
132                 if name in parser.dict['vars']:
133                         address = parser.dict['vars'][name]['count']
134                 elif name in parser.dict['funcs']:
135                         print(f'Error: Identifier is function not array {name}')
136                         parser.success = False
137                 else:
138                         print('Error: Array not declared.')
139                         parser.success = False
140         if parser.success:
141                 p[0] = f'PUSHGP\nPUSHI {address}\nPADD\n{p[3]}LOADN\n'
```

Primeiro, verificamos se o **ID** se encontra no dicionário das variáveis, se sim, empilhamos o valor do **general-pointer** com **PUSHGP**, empilhamos também o valor referente à posição na stack da variável, calculamos a soma destes dois valores, de seguida empilhamos o valor resultante de expressão **(p[3])**, por último, empilhamos o seu valor na stack com **LOADN** no *address* que calculamos com a soma.

4.6 Input-Output

```
def p_stmt1(p):
        'stmt : PRINTI LPAREN exprl RPAREN SC'
        if parser.success:
                p[0] = p[3] + 'WRITEI\n
```

Basta apenas analisar o valor da expressão **(p[3])**, seguido de **WRITEI**, para produzir no output o valor desejado.

4.7 Programas

```
def p_factor9(p):
        'factor : ID LPAREN exprllist RPAREN'
        if parser.success:
                name = p[1]
                args = p[3]
                if name in parser.dict['funcs']:
                        lbl = parser.dict['funcs'][name]['lbl']
                elif name in parser.dict['vars']:
                        print(f'Error: Identifier is variable not function {name}')
                        parser.success = False
                else:
                        print('Error: Function not declared.')
                        parser.success = False
        if parser.success:
                args = args.split('\n')
                res_args = ''
                for arg in args:
                        res_args += f'{arg}\n'
                res_args = res_args[:-1]
                for arg in reversed(parser.dict['funcs'][name]['arguments']):
                        if arg not in ['', '\n']:
                                address = parser.dict['vars'][arg]['count']
                                res_args += f'STOREG {address}\n'
        if parser.success:
                if len(res_args) > 0:
                        p[0] = res_args + f'PUSHA lbl{lbl}\nCALL\n'
                else:
                        p[0] = f'\nPUSHA lbl{lbl}\nCALL\n'
```

Primeiro verificamos se a função foi previamente declarada, caso tenha sido, verificamos se a função recebe argumentos ou não, e, caso tenha, para cada uma, calculamos a sua posição na stack em **address** e guardamos o valor da variável com **STOREG address**. Por último, obtemos o **label** associado á respetiva função, guardada no dicionário das funções, e chamamos a mesma, com **PUSHA label**, seguido de, **CALL**.

Capítulo 5

Codificação e Testes

5.1 Alternativas, Decisões e Problemas de Implementação

De maneira a abringir ao máximo os requisitos pedidos no enunciado, consideramos que a nossa gramática aceitasse inteiros, arrays de uma e duas dimensões de inteiros e strings. Resolvemos ainda que fosse possível declarar funções que retornam valores atómicos, também podendo estas trazer argumentos, e que todas as variáveis têm de ser declarados no inicio do programa, mesmo as utilizadas em funções. Ao nível de operações de controlo de fluxo, optamos para que a linguagem permita o uso do **if-then**, **if-then-else**, e do ciclo **while-do**.

5.2 Testes realizados e Resultados

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

I. Raiz quadrada de um inteiro

```
1  int L;
2  int R;
3  int M;
4  int res;
5  int x;
6
7  def sqrt(x) int {
8      R = x+1;
9      while (L != R - 1)
10     {
11         M = (L + R) / 2;
12
13                 if (M * M <= x)
14                         L = M;
15                 else
16                         R = M;
17         }
18     return L;
19 }
20
21 res = sqrt(atoi(input()));
22 printi(res);
```

Código gerado:

```
1  PUSHI 0
2  PUSHI 0
3  PUSHI 0
4  PUSHI 0
5  PUSHI 0
6  JUMP lbl5
7  lbl4:
8  PUSHG 4
9  PUSHI 1
10 ADD
11 STOREG 1
12 lbl2:
13 PUSHG 0
14 PUSHG 1
15 PUSHI 1
16 SUB
17 EQUAL
18 NOT
19 JZ lbl3
20 PUSHG 0
21 PUSHG 1
22 ADD
23 PUSHI 2
24 DIV
25 FTOI
26 STOREG 2
27 PUSHG 2
28 PUSHG 2
29 MUL
30 PUSHG 4
31 INFEQ
32 JZ lbl0
33 PUSHG 2
34 STOREG 0
35 JUMP lbl1
36 lbl0:
37 PUSHG 2
38 STOREG 1
39 lbl1:
40 JUMP lbl2
41 lbl3:
42 PUSHG 0
43 RETURN
44 lbl5:
45 READ
46 ATOI
47 STOREG 4
48 PUSHA lbl4
49 CALL
50 STOREG 3
51 PUSHG 3
52 WRITEI
```

```
53 STOP
```

II. Sequência de Fibonacci

```
int p1;
int p2;
int t;
int n;

def fib(n) int {
    p1 = 0;
    p2 = 1;
    while (n > 0) {
      t = p1+p2;
      p1 = p2;
      p2 = t;
      n = n-1;
    }
    return p1;
}

printi(fib(atoi(input())));
```

Código gerado:

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
JUMP lbl3
lbl2:
PUSHI 0
STOREG 0
PUSHI 1
STOREG 1
lbl0:
PUSHG 3
PUSHI 0
SUP
JZ lbl1
PUSHG 0
PUSHG 1
ADD
STOREG 2
PUSHG 1
STOREG 0
PUSHG 2
STOREG 1
PUSHG 3
PUSHI 1
SUB
STOREG 3
JUMP lbl0
```

```
29 lbl1:
30 PUSHG 0
31 RETURN
32 lbl3:
33 READ
34 ATOI
35 STOREG 3
36 PUSHA lbl2
37 CALL
38 WRITEI
39 STOP
```

III. Ciclo While-do

```
1  int a;
2  int b;
3  int i;
4
5  i = 0;
6  a = 2;
7  b = 3;
8
9  while (a < 3*b){
10      i = i + 1;
11      a = a + b;
12 }
13
14 printi(a);
```

Código gerado:

```
1  PUSHI 0
2  PUSHI 0
3  PUSHI 0
4  PUSHI 0
5  STOREG 2
6  PUSHI 2
7  STOREG 0
8  PUSHI 3
9  STOREG 1
10 lbl0:
11 PUSHG 0
12 PUSHI 3
13 PUSHG 1
14 MUL
15 INF
16 JZ lbl1
17 PUSHG 2
18 PUSHI 1
19 ADD
20 STOREG 2
21 PUSHG 0
22 PUSHG 1
```

```
23 ADD
24 STOREG 0
25 JUMP lbl0
26 lbl1:
27 PUSHG 0
28 WRITEI
29 STOP
```

IV. Multiplicação de todos os valores de uma matriz por uma constante e impressão dos mesmos

```
1  int mat[2][2];
2  int coe;
3  int row;
4  int col;
5  int i;
6  int j;
7  int rowlen;
8  int collen;
9
10 def mult_matrix (mat, coe, i, j, rowlen, collen) void {
11     while (i < rowlen) {
12         while (j < collen) {
13             mat[i][j] = mat[i][j] * coe;
14             j = j+1;
15         }
16         j = 0;
17         i = i+1;
18     }
19     return;
20 }
21
22 def print_matrix (mat, coe, i, j, rowlen, collen) void {
23     while (i < rowlen) {
24         while (j < collen) {
25             printi(mat[i][j]);
26             println();
27             j = j+1;
28         }
29         j = 0;
30         i = i+1;
31     }
32     return;
33 }
34
35 i = 0;
36 j = 0;
37 coe = 2;
38 rowlen = 2;
39 collen = rowlen;
40
41 mat[0][0] = 1;
42 mat[0][1] = 2;
43 mat[1][0] = 3;
44 mat[1][1] = 4;
```

```
45
46 mult_matrix(mat, coe, i, j, rowlen, collen);
47
48 i = 0;
49 j = 0;
50
51 print_matrix(mat, coe, i, j, rowlen, collen);
```

Código gerado:

```
1  PUSHN 4
2  PUSHI 0
3  PUSHI 0
4  PUSHI 0
5  PUSHI 0
6  PUSHI 0
7  PUSHI 0
8  PUSHI 0
9  JUMP lbl5
10 lbl4:
11 lbl2:
12 PUSHG 7
13 PUSHG 9
14 INF
15 JZ lbl3
16 lbl0:
17 PUSHG 8
18 PUSHG 10
19 INF
20 JZ lbl1
21 PUSHGP
22 PUSHI 0
23 PUSHG 7
24 PUSHI 2
25 MUL
26 ADD
27 PADD
28 PUSHG 8
29 PUSHGP
30 PUSHI 0
31 PUSHG 7
32 PUSHI 2
33 MUL
34 ADD
35 PADD
36 PUSHG 8
37 LOADN
38 PUSHG 4
39 MUL
40 STOREN
41 PUSHG 8
42 PUSHI 1
43 ADD
```

```
44 STOREG 8
45 JUMP lbl0
46 lbl1:
47 PUSHI 0
48 STOREG 8
49 PUSHG 7
50 PUSHI 1
51 ADD
52 STOREG 7
53 JUMP lbl2
54 lbl3:
55 RETURN
56 lbl5:
57 JUMP lbl11
58 lbl10:
59 lbl8:
60 PUSHG 7
61 PUSHG 9
62 INF
63 JZ lbl9
64 lbl6:
65 PUSHG 8
66 PUSHG 10
67 INF
68 JZ lbl7
69 PUSHGP
70 PUSHI 0
71 PUSHG 7
72 PUSHI 2
73 MUL
74 ADD
75 PADD
76 PUSHG 8
77 LOADN
78 WRITEI
79 WRITELN
80 PUSHG 8
81 PUSHI 1
82 ADD
83 STOREG 8
84 JUMP lbl6
85 lbl7:
86 PUSHI 0
87 STOREG 8
88 PUSHG 7
89 PUSHI 1
90 ADD
91 STOREG 7
92 JUMP lbl8
93 lbl9:
94 RETURN
95 lbl11:
96 PUSHI 0
97 STOREG 7
```

```
98 PUSHI 0
99 STOREG 8
100 PUSHI 2
101 STOREG 4
102 PUSHI 2
103 STOREG 9
104 PUSHG 9
105 STOREG 10
106 PUSHGP
107 PUSHI 0
108 PUSHI 0
109 PUSHI 2
110 MUL
111 ADD
112 PADD
113 PUSHI 0
114 PUSHI 1
115 STOREN
116 PUSHGP
117 PUSHI 0
118 PUSHI 0
119 PUSHI 2
120 MUL
121 ADD
122 PADD
123 PUSHI 1
124 PUSHI 2
125 STOREN
126 PUSHGP
127 PUSHI 0
128 PUSHI 1
129 PUSHI 2
130 MUL
131 ADD
132 PADD
133 PUSHI 0
134 PUSHI 3
135 STOREN
136 PUSHGP
137 PUSHI 0
138 PUSHI 1
139 PUSHI 2
140 MUL
141 ADD
142 PADD
143 PUSHI 1
144 PUSHI 4
145 STOREN
146 PUSHG 0
147 PUSHG 4
148 PUSHG 7
149 PUSHG 8
150 PUSHG 9
151 PUSHG 10
```

```
152 STOREG 10
153 STOREG 9
154 STOREG 8
155 STOREG 7
156 STOREG 4
157 STOREG 0
158 PUSHA lbl4
159 CALL
160 PUSHI 0
161 STOREG 7
162 PUSHI 0
163 STOREG 8
164 PUSHG 0
165 PUSHG 4
166 PUSHG 7
167 PUSHG 8
168 PUSHG 9
169 PUSHG 10
170 STOREG 10
171 STOREG 9
172 STOREG 8
173 STOREG 7
174 STOREG 4
175 STOREG 0
176 PUSHA lbl10
177 CALL
178 STOP
```

Veremos agora alguns exemplos em que o ficheiro de entrada tem algum tipo de problema e qual é o resultado que obtemos.

V. Duas variáveis com o mesmo nome

```
1 int a;
2 int a;
```

Como é óbvio não se pode declarar duas variáveis com o mesmo identificador, obtemos então o seguinte resultado:

```
1 Error: Cannot redeclare identifier a.
2 None
```

O mesmo se aplica a arrays, matrizes e funções.

VI. Função com o mesmo identificador que uma variável

```
1 int a;
2
3 def a() void {
4         prints("Errado");
5         return;
6 }
```

Vemos então que não se pode declarar uma função com o mesmo identificador que uma variável.

```
Error: Identifier already declared as variable a
None
```

VII. Chamada de função não definida

```
int a;

def a1() void {
        prints("Errado");
        return;
}

a2();
```

Não se pode chamar uma função que não está definida.

```
Error: Function not declared.
None
```

Capítulo 6

Conclusão

Este projeto trouxe-nos um bom desafio, pois obrigou a que fossem aplicados todos os conhecimentos que fomos aprendendo ao longo do semestre. Apesar de a ideia ser simples (criar uma linguagem de programação e um compilador para a mesma), o projeto mostrou-se desafiante de uma forma positiva.

Olhando para os objetivos e funcionalidades propostas, conseguimos cumprir tudo e até em alguns pequenos casos superar.

Em suma, o nosso conhecimento subiu consideravelmente com o desenvolver do projeto.

6.1 Trabalho Futuro

Apesar de termos cumprido todos os objetivos, existem alguns pontos que poderíamos resolver/adicionar no futuro. O nosso tratamento de erros apesar de existente não é exaustivo e existem casos em que o utilizador não recebe uma mensagem de erro feita por nós mas sim do Python, algo que não deveria acontecer.

Para além disso também seria interessante implementar localidade de variáveis, algo que seria muito interessante pois daria uma maior sensação de que a linguagem de programação que criamos é real.

Simplificar algumas das nossas funções também é, sem dúvida, algo que deveríamos fazer no futuro.

Apêndice A

Código Análise Léxica

```
1  import ply.lex as lex
2  import sys
3  import re
4
5  reserved = {
6          'while': 'WHILE',
7          'if': 'IF',
8          'else': 'ELSE',
9          'def': 'DEF',
10         'int': 'INT',
11         'true': 'TRUE',
12         'false': 'FALSE',
13         'println': 'PRINTLN',
14         'printi': 'PRINTI',
15         'prints': 'PRINTS',
16         'atoi': 'ATOI',
17         'string': 'STRING',
18         'return': 'RETURN',
19         'void': 'VOID',
20         'input': 'INPUT'
21 }
22
23 tokens = (
24         'ID',
25         'ASSIGN',
26         'NUM',
27         'NOT',
28         'NEG',
29         'ADD',
30         'SUBT',
31         'MULT',
32         'DIV',
33         'EQ',
34         'GEQ',
35         'LEQ',
36         'LT',
37         'GT',
38         'NEQ',
39         'AND',
```

```
40              'OR',
41              'SC',
42              'LPAREN',
43              'RPAREN',
44              'LCURLY',
45              'RCURLY',
46              'LBRACKET',
47              'RBRACKET',
48              'COMMA'
49  ) + tuple(reserved.values())
50
51
52  t_ADD = r'\+'
53  t_SUBT = r'\-'
54  t_MULT = r'\*'
55  t_DIV = r'\/'
56
57  t_NOT = r'\!'
58  t_EQ = r'\=\='
59  t_GEQ = r'\>\='
60  t_LEQ = r'\<\='
61  t_LT = r'\<'
62  t_GT = r'\>'
63  t_NEQ = r'\!\='
64  t_AND = r'\&\&'
65  t_OR = r'\|\|'
66
67  t_ASSIGN = r'\='
68
69  t_SC = r';'
70  t_LPAREN = r'\('
71  t_RPAREN = r'\)'
72  t_LCURLY = r'\{'
73  t_RCURLY = r'\}'
74  t_LBRACKET = r'\['
75  t_RBRACKET = r'\]'
76  t_COMMA = r'\,'
77
78  t_ignore = '\t'
79
80  def t_ID(t):
81          r'[a-zA-Z][a-zA-Z0-9_]*'
82          t.type = reserved.get(t.value, 'ID')
83          return t
84
85  def t_NUM(t):
86          r'\d+'
87          t.value = int(t.value)
88          return t
89
90  def t_STRING(t):
91          r'\".*\"'
92          t.type = reserved.get(t.value, 'STRING')
93          return t
```

```
94
95 def t_COMMENT(t):
96         r'\/\/.*'
97         pass
98
99 def t_newline(t):
100     r'\n+'
101     t.lexer.lineno += len(t.value)
102
103 def t_error(t):
104         print(f"Caracter ilegal {t.value[0]}")
105         t.lexer.skip(1)
106
107 lexer = lex.lex()
```

Apêndice B

Código Parser e Tradutor

```
import re
import sys
from ply import yacc
from lexer import tokens

def p_program(p):
        'program : declist funlist codeblock'
        if parser.success:
                p[0] = p[1] + p[2] + p[3] + 'STOP'

def p_codeblock1(p):
        'codeblock : stmlist'
        if parser.success:
                p[0] = p[1]

def p_codeblock2(p):
        'codeblock : '
        if parser.success:
                p[0] = ''

def p_declist_1(p):
        'declist : dec declist'
        if parser.success:
                p[0] = p[1] + p[2]

def p_declist_2(p):
        'declist : '
        if parser.success:
                p[0] = ''

def p_dec1(p):
        'dec : dec_int'
        if parser.success:
                p[0] = p[1]

def p_dec2(p):
        'dec : dec_arr'
        if parser.success:
                p[0] = p[1]
```

```
40
41 def p_dec3(p):
42         'dec : dec_mat'
43         if parser.success:
44                 p[0] = p[1]
45
46 def p_dec_int(p):
47         'dec_int : INT ID SC'
48         if parser.success:
49                 name = p[2]
50                 if name in parser.dict['funcs']:
51                         print(f'Error: Identifier already declared as function {name}.')
52                         parser.success = False
53                 elif name in parser.dict['vars']:
54                         print(f'Error: Identifier already declared as variable {name}')
55                         parser.success = False
56         if parser.success:
57                 parser.dict['vars'].update({name:
58                         {'size': 0,
59                          'count': parser.count
60                         }
61                  }
62                 )
63                 parser.count += 1
64         p[0] = 'PUSHI 0\n'
65
66 def p_def_arr(p):
67         'dec_arr : INT ID LBRACKET NUM RBRACKET SC'
68         if parser.success:
69                 name = p[2]
70                 size = p[4]
71                 if name in parser.dict['funcs']:
72                         print(f'Error: Identifier already declared as function {name}.')
73                         parser.success = False
74                 elif name in parser.dict['vars']:
75                         print(f'Error: Identifier already declared as variable {name}')
76                         parser.success = False
77         if parser.success:
78                 parser.dict['vars'].update({name:
79                         {'size': size,
80                          'count': parser.count
81                         }
82                 }
83                 )
84                 parser.count += size
85         p[0] = f'PUSHN {size}\n'
86
87 def p_def_mat(p):
88         'dec_mat : INT ID LBRACKET NUM RBRACKET LBRACKET NUM RBRACKET SC'
89         if parser.success:
```

```
90                  name = p[2]
91                  row = p[4]
92                  col = p[7]
93                  if name in parser.dict['funcs']:
94                          print(f'Error: Identifier already declared as function {name}.')
95                          parser.success = False
96                  elif name in parser.dict['vars']:
97                          print(f'Error: Identifier already declared as variable {name}')
98                          parser.success = False
99          if parser.success:
100                 parser.dict['vars'].update({name:
101                         {'size': row*col,
102                          'count': parser.count,
103                          'row':row,
104                          'col':col
105                         }
106                 }
107                 )
108                 parser.count += row*col
109         p[0] = f'PUSHN {row*col}\n'
110
111 def p_funlist1(p):
112         'funlist : fun funlist'
113         if parser.success:
114                 p[0] = p[1] + p[2]
115
116 def p_funlist2(p):
117         'funlist : '
118         if parser.success:
119                 p[0] = ''
120
121 def p_fun1(p):
122         'fun : DEF ID LPAREN idlist RPAREN INT LCURLY stmlist RETURN exprl SC RCURLY'
123         name = p[2]
124         if parser.success:
125                 if name in parser.dict['funcs']:
126                         print(f'Error: Cannot redeclare function {name}')
127                         parser.success = False
128                 if name in parser.dict['vars']:
129                         print(f'Error: Identifier already declared as variable {name}')
130                         parser.success = False
131         if parser.success:
132                 lbl = parser.label
133                 lbl_next = parser.label + 1
134                 parser.label += 2
135                 arguments = p[4].split('\n')
136                 parser.dict['funcs'].update({name:
137                         {'name': p[2],
138                          'arguments': arguments,
139                          'statements': p[8],
140                          'lbl': lbl
```

```
                        }
                }
                )
        if parser.success:
                p[0] = f'JUMP lbl{lbl_next}\nlbl{lbl}:\n{p[8]}{p[10]}RETURN\nlbl{
                    lbl_next}:\n'

def p_fun2(p):
        'fun : DEF ID LPAREN idlist RPAREN VOID LCURLY stmlist RETURN SC RCURLY'
        name = p[2]
        if parser.success:
                if name in parser.dict['funcs']:
                        print(f'Error: Cannot redeclare function {name}')
                        parser.success = False
                if name in parser.dict['vars']:
                        print(f'Error: Identifier already declared as variable {name}
                            ')
                        parser.success = False
        if parser.success:
                lbl = parser.label
                lbl_next = parser.label + 1
                parser.label += 2
                arguments = p[4].split('\n')
                parser.dict['funcs'].update({name:
                        {'name': p[2],
                         'arguments': arguments,
                         'statements': p[8],
                         'lbl': lbl
                        }
                }
                )
                p[0] = f'JUMP lbl{lbl_next}\nlbl{lbl}:\n{p[8]}RETURN\nlbl{lbl_next}:\
                    n'

def p_idlist1(p):
        'idlist : ID cont'
        if parser.success:
                p[0] = p[1] + '\n' + p[2]

def p_idlist2(p):
        'idlist : '
        if parser.success:
                p[0] = ''

def p_cont1(p):
        'cont : COMMA ID cont'
        if parser.success:
                p[0] = p[2] + '\n' + p[3]

def p_cont2(p):
        'cont : '
        if parser.success:
                p[0] = ''

```

```
192 def p_stmlist1(p):
193         'stmlist : stmt stmlist'
194         if parser.success:
195                 p[0] = p[1] + p[2]
196
197 def p_stmlist2(p):
198         'stmlist : stmt'
199         if parser.success:
200                 p[0] = p[1]
201
202 def p_stmt1(p):
203         'stmt : PRINTI LPAREN exprl RPAREN SC'
204         if parser.success:
205                 p[0] = p[3] + 'WRITEI\n'
206
207 def p_stmt2(p):
208         'stmt : PRINTLN LPAREN RPAREN SC'
209         if parser.success:
210                 p[0] = 'WRITELN\n'
211
212 def p_stmt3(p):
213         'stmt : PRINTS LPAREN STRING RPAREN SC'
214         if parser.success:
215                 p[0] = f'PUSHS {p[3]}\nWRITES\n'
216
217 def p_stmt4(p):
218         'stmt : WHILE LPAREN exprl RPAREN block'
219         if parser.success:
220                 lbl_ini = parser.label
221                 lbl_end = parser.label + 1
222                 parser.label += 2
223                 p[0] = f'lbl{lbl_ini}:\n{p[3]}JZ lbl{lbl_end}\n{p[5]}JUMP lbl{lbl_ini
                        }\nlbl{lbl_end}:\n'
224
225 def p_stmt5(p):
226         'stmt : ID ASSIGN exprl SC'
227         if parser.success:
228                 name = p[1]
229                 if name in parser.dict['vars']:
230                         address = parser.dict['vars'][name]['count']
231                 elif name in parser.dict['funcs']:
232                         print(f'Error: Identifier is function not variable {name}')
233                         parser.success = False
234                 else:
235                         print('Error: Variable not declared.')
236                         parser.success = False
237         if parser.success:
238                 p[0] = f'{p[3]}STOREG {address}\n'
239
240 def p_stmt6(p):
241         'stmt : ID LBRACKET exprl RBRACKET ASSIGN exprl SC'
242         if parser.success:
243                 name = p[1]
244                 if name in parser.dict['vars']:
```

```
245                         address = parser.dict['vars'][name]['count']
246                 elif name in parser.dict['funcs']:
247                         print(f'Error: Identifier is function not array {name}')
248                         parser.success = False
249                 else:
250                         print('Error: Array not declared.')
251                         parser.success = False
252         if parser.success:
253                 p[0] = f'PUSHGP\nPUSHI {address}\nPADD\n{p[3]}{p[6]}STOREN\n'
254
255 def p_stmt7(p):
256         'stmt : ID LBRACKET exprl RBRACKET LBRACKET exprl RBRACKET ASSIGN exprl SC'
257         if parser.success:
258                 name = p[1]
259                 row = p[3]
260                 col = p[6]
261                 if name in parser.dict['vars']:
262                         address = parser.dict['vars'][name]['count']
263                         tot_col = parser.dict['vars'][name]['col']
264                 elif name in parser.dict['funcs']:
265                         print(f'Error: Identifier is function not matrix {name}')
266                         parser.success = False
267                 else:
268                         print('Error: Matrix not declared.')
269                         parser.success = False
270                 if parser.success:
271                         p[0] = f'PUSHGP\nPUSHI {address}\n{row}PUSHI {tot_col}\nMUL\
                                nADD\nPADD\n{col}{p[9]}STOREN\n'
272
273 def p_stmt8(p):
274         'stmt : IF LPAREN exprl RPAREN block ELSE block'
275         if parser.success:
276                 lbl_else = parser.label
277                 lbl_end = parser.label + 1
278                 parser.label += 2
279                 p[0] = f'{p[3]}JZ lbl{lbl_else}\n{p[5]}JUMP lbl{lbl_end}\nlbl{
                        lbl_else}:\n{p[7]}lbl{lbl_end}:\n'
280
281 def p_stmt9(p):
282         'stmt : IF LPAREN exprl RPAREN block'
283         if parser.success:
284                 lbl_end = parser.label
285                 parser.label += 1
286                 p[0] = f'{p[3]}JZ lbl{lbl_end}\n{p[5]}lbl{lbl_end}:\n'
287
288 def p_stmt10(p):
289         'stmt : INPUT LPAREN RPAREN SC'
290         p[0] = 'READ\n'
291
292 def p_stmt11(p):
293         'stmt : exprl SC'
294         p[0] = f'{p[1]}'
295
296 def p_block1(p):
```

```
297         'block : LCURLY stmlist RCURLY'
298         if parser.success:
299                 p[0] = p[2]
300
301 def p_block2(p):
302         'block : LCURLY stmt RCURLY'
303         if parser.success:
304                 p[0] = p[2]
305
306 def p_block3(p):
307         'block : stmt'
308         if parser.success:
309                 p[0] = p[1]
310
311 def p_exprl1(p):
312         'exprl : expr oprl exprl'
313         if parser.success:
314                 p[0] = p[1] + p[3] + p[2]
315
316 def p_exprl2(p):
317         'exprl : expr'
318         if parser.success:
319                 p[0] = p[1]
320
321 def p_expr1(p):
322         'expr : expr opra term'
323         if parser.success:
324                 p[0] = p[1] + p[3] + p[2]
325
326 def p_expr2(p):
327         'expr : term'
328         if parser.success:
329                 p[0] = p[1]
330
331 def p_term1(p):
332         'term : term oprm factor'
333         if parser.success:
334                 p[0] = p[1] + p[3] + p[2]
335
336 def p_term2(p):
337         'term : factor'
338         if parser.success:
339                 p[0] = p[1]
340
341 def p_factor1(p):
342         'factor : LPAREN expr RPAREN'
343         if parser.success:
344                 p[0] = p[2]
345
346 def p_factor2(p):
347         'factor : ID'
348         if parser.success:
349                 name = p[1]
350                 if name in parser.dict['vars']:
```

```
351                         address = parser.dict['vars'][name]['count']
352                 elif name in parser.dict['funcs']:
353                         print(f'Error: Identifier is function not variable {name}')
354                         parser.success = False
355                 else:
356                         print('Error: Variable not declared.')
357                         parser.success = False
358         if parser.success:
359                 p[0] = f'PUSHG {address}\n'
360
361 def p_factor3(p):
362         'factor : NUM'
363         if parser.success:
364                 p[0] = f'PUSHI {p[1]}\n'
365
366 def p_factor4(p):
367         'factor : NOT exprl'
368         if parser.success:
369                 p[0] = p[2] + 'NOT\n'
370
371 def p_factor5(p):
372         'factor : NEG exprl'
373         if parser.success:
374                 p[0] = 'PUSHI 0\n' + p[2] + 'SUB\n'
375
376 def p_factor6(p):
377         'factor : ID LBRACKET exprl RBRACKET'
378         if parser.success:
379                 name = p[1]
380                 if name in parser.dict['vars']:
381                         address = parser.dict['vars'][name]['count']
382                 elif name in parser.dict['funcs']:
383                         print(f'Error: Identifier is function not array {name}')
384                         parser.success = False
385                 else:
386                         print('Error: Array not declared.')
387                         parser.success = False
388         if parser.success:
389                 p[0] = f'PUSHGP\nPUSHI {address}\nPADD\n{p[3]}LOADN\n'
390
391 def p_factor7(p):
392         'factor : ID LBRACKET exprl RBRACKET LBRACKET exprl RBRACKET'
393         if parser.success:
394                 name = p[1]
395                 row = p[3]
396                 col = p[6]
397                 if name in parser.dict['vars']:
398                         address = parser.dict['vars'][name]['count']
399                         tot_col = parser.dict['vars'][name]['col']
400                 elif name in parser.dict['funcs']:
401                         print(f'Error: Identifier is function not matrix {name}')
402                         parser.success = False
403                 else:
404                         print('Error: Matrix not defined.')
```

```
405                         parser.success = False
406         if parser.success:
407                 p[0] = f'PUSHGP\nPUSHI {address}\n{row}PUSHI {tot_col}\nMUL\nADD\
                        nPADD\n{col}LOADN\n'
408
409 def p_factor8(p):
410         'factor : ATOI LPAREN argatoi RPAREN'
411         if parser.success:
412                 p[0] = f'{p[3]}\nATOI\n'
413
414 def p_factor9(p):
415         'factor : ID LPAREN exprllist RPAREN'
416         if parser.success:
417                 name = p[1]
418                 args = p[3]
419                 if name in parser.dict['funcs']:
420                         lbl = parser.dict['funcs'][name]['lbl']
421                 elif name in parser.dict['vars']:
422                         print(f'Error: Identifier is variable not function {name}')
423                         parser.success = False
424                 else:
425                         print('Error: Function not declared.')
426                         parser.success = False
427         if parser.success:
428                 args = args.split('\n')
429                 res_args = ''
430                 for arg in args:
431                         res_args += f'{arg}\n'
432                 res_args = res_args[:-1]
433                 for arg in reversed(parser.dict['funcs'][name]['arguments']):
434                         if arg not in ['', '\n']:
435                                 address = parser.dict['vars'][arg]['count']
436                                 res_args += f'STOREG {address}\n'
437         if parser.success:
438                 if len(res_args) > 0:
439                         p[0] = res_args + f'PUSHA lbl{lbl}\nCALL\n'
440                 else:
441                         p[0] = f'\nPUSHA lbl{lbl}\nCALL\n'
442
443 def p_factor10(p):
444         'factor : TRUE'
445         p[0] = 'PUSHI 1\n'
446
447 def p_factor11(p):
448         'factor : FALSE'
449         p[0] = 'PUSHI 0\n'
450
451 def p_exprllist1(p):
452         'exprllist : exprl contexprllist'
453         p[0] = p[1] + p[2]
454
455 def p_exprllist2(p):
456         'exprllist : '
457         p[0] = ''
```

```
458
459 def p_contexprllist(p):
460         'contexprllist : COMMA exprl contexprllist'
461         p[0] = p[2] + p[3]
462
463 def p_contexprllist2(p):
464         'contexprllist : '
465         p[0] = ''
466
467 def p_argatoi1(p):
468         'argatoi : STRING'
469         if parser.success:
470                 p[0] = f'PUSHS {p[1]}'
471
472 def p_argatoi2(p):
473         'argatoi : INPUT LPAREN RPAREN'
474         if parser.success:
475                 p[0] = f'READ'
476
477 def p_opra1(p):
478         'opra : ADD'
479         if parser.success:
480                 p[0] = 'ADD\n'
481
482 def p_opra2(p):
483         'opra : SUBT'
484         if parser.success:
485                 p[0] = 'SUB\n'
486
487 def p_oprm1(p):
488         'oprm : MULT'
489         if parser.success:
490                 p[0] = 'MUL\n'
491
492 def p_oprm2(p):
493         'oprm : DIV'
494         if parser.success:
495                 p[0] = 'DIV\nFTOI\n'
496
497 def p_oprl1(p):
498         'oprl : EQ'
499         if parser.success:
500                 p[0] = 'EQUAL\n'
501
502 def p_oprl2(p):
503         'oprl : GEQ'
504         if parser.success:
505                 p[0] = 'SUPEQ\n'
506
507 def p_oprl3(p):
508         'oprl : LEQ'
509         if parser.success:
510                 p[0] = 'INFEQ\n'
511
```

```
512 def p_oprl4(p):
513         'oprl : LT'
514         if parser.success:
515                 p[0] = 'INF\n'
516
517 def p_oprl5(p):
518         'oprl : GT'
519         if parser.success:
520                 p[0] = 'SUP\n'
521
522 def p_oprl6(p):
523         'oprl : NEQ'
524         if parser.success:
525                 p[0] = 'EQUAL\nNOT\n'
526
527 def p_oprl7(p):
528         'oprl : AND'
529         if parser.success:
530                 p[0] = 'AND\n'
531
532 def p_oprl8(p):
533         'oprl : OR'
534         if parser.success:
535                 p[0] = 'OR\n'
536
537 def p_error(p):
538         parser.success = False
539         print(f'Error: Error in line {parser.line}\n{p}')
540
541 parser = yacc.yacc(start='program')
542 parser.count = 0
543 parser.label = 0
544 parser.line = 0
545
546 parser.dict = {
547         'vars': {},
548         'funcs': {}
549 }
550
551 if __name__ == '__main__':
552         file = sys.argv[1]
553         f = open(file, 'r+')
554         parser.success = True
555         res = parser.parse(f.read())
556         print(res)
557         f.close()
```