Relatório de Desenvolvimento

Alexis Castro Correia (A102495)
João de Albuquerque Ferreira Vieira da Fonseca (A102512)

11 de janeiro de 2025

Resumo

Este relatório descreve o desenvolvimento de uma linguagem de programação imperativa simples, juntamente com seu compilador correspondente, conforme especificado na declaração do projeto. A linguagem foi projetada para facilitar a declaração de variáveis atômicas inteiras, a execução de operações aritméticas, relacionais e lógicas, bem como instruções de fluxo de controle (seleção e iteração). Além disso, recursos adicionais foram implementados, mais especificamente, a definição e invocação de subprogramas sem parâmetros.

O compilador, criado com o auxílio das ferramentas PLY/Python, traduz o código-fonte escrito nesta linguagem em pseudocódigo Assembly para execução em uma Máquina Virtual (EWVM). Este documento engloba a descrição do processo de implementação, exemplos de código na nova linguagem e validação através testes.

Conteúdo

1 Introdução

Este relatório descreve o processo de concepção e implementação de uma linguagem de programação imperativa simples, bem como o desenvolvimento de um compilador para traduzir programas escritos nessa linguagem para Assembly de uma máquina virtual (EWVM). Este trabalho foi realizado no contexto da disciplina *Processamento de linguagens e compiladores*, com o objetivo de consolidar os conhecimentos sobre gramáticas formais, construção de compiladores e geração de código Assembly.

A linguagem foi projetada para suportar as principais funcionalidades de linguagens de programação imperativas, incluindo a declaração de variáveis atômicas, manipulação de estruturas de controle de fluxo, execução de operações aritméticas e lógicas, leitura e escrita em *standard input/output*, e a implementação de ciclos. Além disso, foram incorporadas funcionalidades opcionais, como o suporte a *arrays* ou subprogramas, de acordo com os requisitos estabelecidos no enunciado do projeto.

O relatório está organizado da seguinte forma: inicialmente, apresenta-se o contexto e os requisitos para a definição da linguagem e do compilador. Em seguida, detalha-se o processo de construção da gramática independente de contexto (GIC) e a utilização de ferramentas como *Ply.Lex* e *Ply.Yacc*. Posteriormente, discute-se a implementação do compilador, incluindo a geração de código Assembly e a execução em uma máquina virtual. Por fim, são apresentados os testes realizados, a análise dos resultados obtidos e as conclusões gerais sobre o trabalho.

Este projeto oferece uma experiência prática em todas as etapas de desenvolvimento de uma linguagem de programação e de um compilador, reforçando conceitos fundamentais da teoria da computação e da engenharia de software.

2 Contextualização

2.1 Descrição do Problema

O objetivo deste projeto é criar e implementar instruções simples que permitam aos programadores realizar tarefas simples, como declarar variáveis, executar código, controlar o fluxo de opções e padrões de iteração e ler/escrever dados. Além disso, a linguagem deve incluir recursos adicionais, como suporte para matrizes ou sub-rotinas, para fornecer uma base sólida para a resolução de problemas básicos de computador. É um pseudocódigo e um código assembly que permite que ele seja executado em uma máquina virtual. Portanto, o principal problema é criar um sistema eficaz e eficiente que atenda ao processo de ensino específico e leve em consideração as limitações e características do idioma.

2.2 Linguagem

O nosso grupo decidiu escolher a linguagem C para tradução e compilação neste projeto porque é simples, familiar e atende bem às necessidades do enunciado. A sintaxe da linguagem é simples e fácil de entender, tornando a definição da sintaxe e o compilador fáceis de usar. Além disso, C já é uma linguagem que suporta muitos conceitos como declaração de variáveis, operações aritméticas e lógicas, expressões de controle de fluxo como "if", "while", "for", que são importantes para a resolução de problemas. Assim, estar familiarizado com C nos ajudará a entender melhor como funciona o processo de conversão de código em código de máquina, o que é importante em nosso trabalho com máquinas virtuais. Resumindo, C fornece uma boa estrutura para processamento simples e eficiente de análise e compilação de linguagem.

3 Análisador Léxico

Para a resolução do problema construímos o analisador léxico seguindos os seguintes passos

```
import ply.lex as lex

literals = ['(' , ')' , '{' , '}', ';' , ',' , '&']

tokens = ('ID', 'INT', 'INTT','STRING', 'ADD', 'SUB',
          'MUL', 'DIV','EQ', 'NEQ', 'LT', 'LE', 'GT',
          'GE', 'WRITE','READ', 'INCLUDE', 'BIBLIO',
          'IF', 'ELSE', 'FOR', 'WHILE','RETURN',
          'COMENT', 'ATRIBUICAO', 'NOT', 'AND', 'OR')

def t_COMENT(t):
    r'//[^\n]*'
    return t

def t_BIBLIO(t):
    r'<[A-z0-9][A-z0-9_-]*\.h>'
    return t

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

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

def t_MUL(t):
    r'\*'
    return t

def t_DIV(t):
    r'/'
    return t

def t_EQ(t):
    r'=='
```

```
37          return t
38
39      def t_NEQ(t):
40          r'\!='
41          return t
42
43      def t_NOT(t):
44          r'\!(?!=)'
45          return t
46
47      def t_LE(t):
48          r'<='
49          return t
50
51      def t_GE(t):
52          r'>='
53          return t
54
55      def t_LT(t):
56          r'<(?!=)'
57          return t
58
59      def t_GT(t):
60          r'>(?!=)'
61          return t
62
63      def t_AND(t):
64          r'&&'
65          return t
66
67      def t_OR(t):
68          r'\|\|'
69          return t
70
71      def t_ATRIBUICAO(t):
72          r'=(?!=)'
73          return t
74
75      def t_INTT(t):
76          r'int'
77          return t
78
79      def t_INCLUDE(t):
80          r'\#[ ]?include'
81          return t
82
83      def t_IF(t):
84          r'if'
```

```
85          return t
86
87      def t_ELSE(t):
88          r'else'
89          return t
90
91      def t_FOR(t):
92          r'for'
93          return t
94
95      def t_WHILE(t):
96          r'while'
97          return t
98
99      def t_RETURN(t):
100         r'return'
101         return t
102
103     def t_WRITE(t):
104         r'printf'
105         return t
106
107     def t_READ(t):
108         r'scanf'
109         return t
110
111     def t_STRING(t):
112         r'\".+\"'
113         return t
114
115     def t_INT(t):
116         r'(-)?[0-9]+(?!\.)'
117         return t
118
119     def t_ID(t):
120         r'[A-z][A-z0-9_]*'
121         return t
122
123     t_ignore = ' \n\t'
124
125     def t_error(t):
126         print('Illegal character: ', t.value[0])
127         t.lexer.skip(1)
128
129     lexer = lex.lex()
```

Primeiramente impou-se a biblioteca que nos permite criar o analisador léxico,seguida de dois conceitos distintos:

Literals → São defenidos caracteres específicos ((,), , , ;, ,,&) que são diretamente reconhecidos como tokens. Esses símbolos são usados frequentemente na linguagem C para delimitação ou controle.

Tokens → São listados todos os tipos de tokens que o analisador léxico precisa identificar no código C. Cada token representa uma unidade significativa na linguagem, como palavras-chave, operadores, literais e identificadores.

ID → Representa identificadores, que são nomes definidos pelo programador para variáveis, funções, ou outros elementos.

INT → Representa números inteiros.

INTT → Representa o palavra-chave "int"que é utilizada na declaração de variáveis e na definição de funções.

STRING → Representa cadeias de caracteres delimitadas por aspas duplas (").

ADD → Representa a operação soma.
SUB → Representa a operação subtração.
MUL → Representa a operação multiplicação.
DIV → Representa a operação divisão.
EQ → Representa a comparação igualdade.
NEQ → Representa a comparação diferença.
LT → Representa a comparação menor que.
LE → Representa a comparação menor ou igual.
GT → Representa a comparação maior que.
GE → Representa a comparação maior igual a.
WRITE → Representa a função padrão (em C) write.
READ → Representa a função padrão (em C) read.
INCLUDE → Indica uma inclusão (equivalente a um "import"no Python).
BIBLIO → Representa bibliotecas padrão (em C).
IF → Representa a estrutura de controle de fluxo if.
ELSE → Representa a estrutura de controle de fluxo else.
FOR → Representa a estrutura de controle de fluxo for.
WHILE → Representa a estrutura de controle de fluxo while.
RETURN → Representa a estrutura de controle de fluxo return.
COMENT → Representa comentários (de linha única) no código
ATRIBUICAO → Representa o operador de atribuição (=).
NOT → Representa negação lógica.
AND → Representa e lógico.
OR → Representa ou lógico.

Com o analisador léxico pronto temos agora uma base sólida para construir um BNF que represente a linguagem C juntamente com o analisador sintático.

4 Análisador Sintático

4.1 BNF

No desenvolvimento do BNF, nosso objetivo foi criar uma representação simplificada e acessível da linguagem C, que fosse clara e fácil de compreender. Com isso em mente, apresentamos o BNF passo a passo com as respetivas descrições.

```
Programa ::= Imports Funcs
```

Representa o programa principal, que consiste em declarações de bibliotecas (Imports) seguidas de funções (Funcs).

```
Imports ::= Import
          | Import Imports
```

Define que o programa pode ter uma ou mais declarações de bibliotecas. No caso da linguagem C, é preciso incluir a biblioteca "stdio.h"para poder ler e escrever (stdin/stdout). Por isso, de acordo com o enunciado, o Imports nunca poderá ser vazio.

```
Import ::= INCLUDE BIBLIO
```

Define a sintaxe para incluir uma biblioteca

```
Funcs ::= Func
        | Func Funcs
```

Representa uma ou mais definições de funções no programa; já que, no mínimo, haverá a função *main*.

```
Func ::= Tipo ID '(' ')' '{' Declarations Lines Output '}'
```

Define a estrutura de uma função: tipo de retorno (Tipo), nome da função (ID), parâmetros (aqui sem parâmetros definidos), bloco da função com declarações (Declarations), linhas de execução (Lines) e saída (Output).

```
Tipo ::= INTT
```

Define o tipo de dados da variável ou do retorno de uma função. Para este trabalho, estaremos apenas utilizando inteiros, por isso, o único tipo que nos interessa é o tipo inteiro. No entanto, a linguagem C ainda permite variaáveis do tipo *char* e *float* e funções também podem não retornar valores (tipo *void*).

```
Declarations ::= €
               | Declaration Declarations
```

Declara variáveis dentro de um bloco de código. Note que € denota o vazio, ou seja, não é obrigatório que haja declaração de variáveis no código.

```
Declaration ::= Tipo VarList ';'
              | Tipo ID ATRIBUICAO Expression ';'
```

Declara uma lista de variáveis do mesmo tipo ou inicia uma variável com um valor.

```
VarList ::= ID
          | ID ',' VarList
```

Representa uma lista de variáveis separadas por vírgula.

```
Expression ::= Expression ADD Expression
             | Expression SUB Expression
             | Expression MUL Expression
             | Expression DIV Expression
             | '(' Expression ')'
             | ID
             | Value
             | Call
```

Representa expressões matemáticas (+,-,*,/) e valores que podem ser atribuídos a uma variável.

```
Value ::= INT
```

Representa valores literais numéricos inteiros. Da mesma forma que o tipo, escrevemos a gramática assim para surportar a adição de outros tipos e valores posteriormente.

```
Call ::= ID '(' ')'
```

Representa a chamada de uma função sem parâmetros por seu identificador (ID).

```
Lines ::= €
        | Line Lines
```

Representa as linhas de execução no corpo de uma função.

```
Line ::= Atribuition
       | Select
       | Cicle
       | Read
       | Write
       | COMENT
```

Representa uma única linha de execução(atribuição, seleção (if-else), ciclo (while/for), leitura (Read), escrita (Write), ou comentário (COMENT)).

```
Atribuition ::= ID ATRIBUICAO Expression ';'
```

Realiza uma atribuição de valor a uma variável.

```
Select ::= IF '(' Conditions ')' '{' Lines '}' Else
```

Define uma estrutura condicional (if-else).

```
Else ::= ELSE '{' Lines '}'
       | €
```

Bloco opcional que executa quando a condição do if é falsa.

```
Cicle ::= WHILE '(' Conditions ')' '{' Lines '}'
        | FOR '(' ID ATRIBUICAO INT ';' Conditions ';' Math ')' '{' Lines
          '}'
```

Define ciclos (loops) no programa : while ou for.

```
Conditions ::= Condition
             | Condition AND Conditions
             | Condition OR Conditions
```

Representa as condições lógicas que serão usadas nas estrutruas de controlo de fluxo.

```
Condition ::= Expression EQ Expression
            | Expression NEQ Expression
            | Expression LT Expression
            | Expression LE Expression
            | Expression GT Expression
            | Expression GE Expression
            | NOT '(' Condition ')'
```

Avalia uma expressão lógica

```
Math ::= Atribuition
       | Atribuition ',' Math
```

Representa operações matemáticas que podem ser realizados no cilo "for".

```
Read ::= READ '(' STRING ',' Addresses ')' ';'
```

Lê valores de entrada e os armazena em variáveis.

```
Addresses ::= Address
            | Address ',' Addresses
```

Representa uma lista de endereços de memória onde os valores serão armazenados.

```
Address ::= '&' ID
```

Representa o endereço de uma variável.

```
Write ::= WRITE '(' STRING ')' ';'
        | WRITE '(' STRING ',' VarList ')' ';'
```

Imprime valores na saída padrão (standar output).

```
Output ::= RETURN Ret ';'
```

Define a instrução de retorno de uma função.

```
Ret ::= ID
      | Value
      | €
```

Define o valor retornado pela função, que pode ser uma variável ou um valor literal (ou serem vazios).

Com isto tudo temos o seguinte BNF.

```
Programa ::= Imports Funcs
Imports ::= Import
          | Import Imports
Import ::= INCLUDE BIBLIO
Funcs ::= Func
        | Func Funcs
Func ::= Tipo ID '(' ')' '{' Declarations Lines Output '}'
Tipo ::= INTT
Declarations ::= €
               | Declaration Declarations
Declaration ::= Tipo VarList ';'
              | Tipo ID ATRIBUICAO Expression ';'
VarList ::= ID
          | ID ',' VarList
Expression ::= Expression ADD Expression
             | Expression SUB Expression
             | Expression MUL Expression
             | Expression DIV Expression
             | '(' Expression ')'
             | ID
             | Value
             | Call
Value ::= INT
Call ::= ID '(' ')'
Lines ::= €
        | Line Lines
Line ::= Atribuition
        | Select
        | Cicle
        | Read
        | Write
        | COMENT
Atribuition ::= ID ATRIBUICAO <Expression> ';'
Select ::= IF '(' Conditions ')' '{' <Lines> '}' Else
Else ::= ELSE '{' Lines '}'
       |
Cicle ::= WHILE '(' Conditions ')' '{' Lines '}'
```

```
        | FOR '(' ID ATRIBUICAO INT ';' Conditions ';' Math ')' '{' Lines
          '}'
Conditions ::= Condition
             | Condition AND Conditions
             | Condition OR Conditions
Condition ::= Expression EQ Expression
             | Expression NEQ Expression
             | Expression LT Expression
             | Expression LE Expression
             | Expression GT Expression
             | Expression GE Expression
             | NOT '(' Condition ')'
Math ::= Atribuition
       | Atribuition ',' Math
Read ::= READ '(' STRING ',' <Addresses> ')' ';'
Addresses ::= Address
            | Address ',' Addresses
Address ::= '&' ID
Write ::= WRITE '(' STRING ')' ';'
        | WRITE '(' STRING ',' VarList ')' ';'
Output ::= RETURN Ret ';'
Ret ::= ID
      | Value
      | €
```

4.2 Máquina Virtual

De acordo com o enunciado do trabalho, foi nos pedido que fosse gerado um código assembly por isso foi necessário acrescentar certas especificações no yacc.

```
    import ply.yacc as yacc

    from plc24TP2gr15_lex import tokens

    def p_Programa(p):
        "Programa : Imports Funcs"

    def p_Imports1(p):
        "Imports : Import"

    def p_Imports2(p):
        "Imports : Import Imports"

    def p_Import(p):
        "Import : INCLUDE BIBLIO"

```

```
17      def p_Funcs1(p):
18          "Funcs : Func"
19
20      def p_Funcs2(p):
21          "Funcs : Func Funcs"
22
23      def p_Func(p):
24          "Func : Tipo ID '(' ')' '{' Declarations Lines Output '}'"
25          o = parser.aux.pop()
26          l = parser.aux.pop()
27          d = parser.aux.pop()
28          f = f"{p[2]}:\n"
29          if p[2] == "main":
30              o = o.replace("RETURN", "STOP")
31              f = f+d+l+o
32          else:
33              f = f+l+o
34          #parser.aux.append(f)
35          parser.mv = parser.mv + f
36          parser.aux.clear()
37
38      def p_Tipo(p):
39          "Tipo : INTT"
40          parser.type.append("PUSHI")
41
42      def p_Declarations1(p):
43          "Declarations : "
44          s = ""
45          for c in parser.aux:
46              s = s + c
47          s = s + "START\n"
48          parser.aux = []
49          parser.aux.append(s)
50          parser.aux.append("AUX")
51          pass
52
53      def p_Declarations2(p):
54          "Declarations : Declaration Declarations"
55
56      def p_Declaration1(p):
57          "Declaration : Tipo VarList ';'"
58          parser.type.pop()
59
60      def p_Declaration2(p):
61          "Declaration : Tipo ID ATRIBUICAO Expression ';'"
62          parser.type.pop()
63          if p[2] not in parser.reg:
64              parser.reg.append(p[2])
```

```
65          else:
66              parser.aux.append(f"ERR \"Variável {p[1]} já declarada\"\n")
67
68      def p_VarList1(p):
69          "VarList : ID "
70          if p[1] not in parser.reg:
71              parser.reg.append(p[1])
72              t = parser.type[-1]
73              parser.aux.append(f"{t} 0 //{p[1]}\n")
74          else:
75              parser.aux.append(f"ERR \"Variável {p[1]} já declarada\"\n")
76
77      def p_VarList2(p):
78          "VarList : ID ',' VarList"
79          if p[1] not in parser.reg:
80              parser.reg.append(p[1])
81              t = parser.type[-1]
82              parser.aux.append(f"{t} 0 //{p[1]}\n")
83          else:
84              parser.aux.append(f"ERR \"Variável {p[1]} já declarada\"\n")
85
86      def p_Expression1(p):
87          "Expression : Expression ADD Expression"
88          b = parser.aux.pop()
89          a = parser.aux.pop()
90          s = a + b + "ADD\n"
91          parser.aux.append(s)
92
93      def p_Expression2(p):
94          "Expression : Expression SUB Expression"
95          b = parser.aux.pop()
96          a = parser.aux.pop()
97          s = a + b + "SUB\n"
98          parser.aux.append(s)
99
100     def p_Expression3(p):
101         "Expression : Expression MUL Expression"
102         b = parser.aux.pop()
103         a = parser.aux.pop()
104         s = a + b + "MUL\n"
105         parser.aux.append(s)
106
107     def p_Expression4(p):
108         "Expression : Expression DIV Expression"
109         b = parser.aux.pop()
110         a = parser.aux.pop()
111         s = a + b + "DIV\n"
112         parser.aux.append(s)
```

```
113
114     def p_Expression5(p):
115         "Expression : '(' Expression ')'"
116
117     def p_Expression6(p):
118         "Expression : ID"
119         if p[1] in parser.reg:
120             s = f"PUSHG {parser.reg.index(p[1])}\n"
121             parser.aux.append(s)
122         else:
123             parser.aux.append("ERR \"Var não declarada\"\n")
124
125     def p_Expression7(p):
126         "Expression : Value"
127
128     def p_Expression8(p):
129         "Expression : Call"
130
131     def p_Value1(p):
132         "Value : INT"
133         t = parser.type[-1]
134         if t == "PUSHI":
135             s = f"{t} {p[1]}\n"
136             parser.aux.append(s)
137         else:
138             parser.aux.append("ERR \"Valor não é inteiro\"\n")
139
140     def p_Call(p):
141         "Call : ID '(' ')'"
142         s = f"PUSHA {p[1]}\nCALL\n"
143         parser.aux.append(s)
144
145     def p_Lines1(p):
146         "Lines : "
147         s = ""
148         c = parser.aux.pop()
149         while c != "COND" and c != "AUX":
150             s = c + s
151             c = parser.aux.pop()
152         parser.aux.append(s)
153         parser.aux.append("AUX")
154         pass
155
156     def p_Lines2(p):
157         "Lines : Line Lines"
158
159     def p_Line1(p):
160         "Line : Atribuition"
```

```
161
162     def p_Line2(p):
163         "Line : Select"
164         parser.c = parser.c + 1
165         if parser.c == parser.C:
166             parser.c = 0
167
168     def p_Line3(p):
169         "Line : Cicle"
170         parser.c = parser.c + 1
171         if parser.c == parser.C:
172             parser.c = 0
173
174     def p_Line4(p):
175         "Line : Read"
176
177     def p_Line5(p):
178         "Line : Write"
179
180     def p_Line6(p):
181         "Line : COMENT"
182         f"{p[1]}\n"
183
184     def p_Atribuition(p):
185         "Atribuition : ID ATRIBUICAO Expression ';'"
186         s = f"STOREG {parser.reg.index(p[1])}\n"
187         parser.aux[-1] = parser.aux[-1] + s
188
189     def p_Select(p):
190         "Select : IF '(' Conditions ')' '{' Lines '}' Else"
191         e = parser.aux.pop()
192         i = parser.aux.pop()
193         c = parser.aux.pop()
194         s = c + "JZ Else\n" + i + f"JUMP End{parser.C-parser.c}\n" + e
195         parser.aux.append(s)
196
197     def p_Else1(p):
198         "Else : ELSE '{' Lines '}'"
199         parser.aux.pop()
200         e = parser.aux.pop()
201         s = "Else: //NOP\n" + e + f"End{parser.C-parser.c}: //NOP\n"
202         parser.aux.append(s)
203
204     def p_Else2(p):
205         "Else : "
206         pass
207
208     def p_Cicle1(p):
```

```
209         "Cicle : WHILE '(' Conditions ')' '{' Lines '}'"
210         parser.aux.pop()
211         cc = parser.aux.pop()
212         c = parser.aux.pop()
213         s = "Flag: //NOP\n" + c + f"JZ End{parser.C-parser.c}:\n" + cc +f"
               JUMP Flag\nEnd{parser.C-parser.c}: //NOP\n"
214         parser.aux.append(s)
215
216     def p_Cicle2(p):
217         "Cicle : FOR '(' ID ATRIBUICAO INT ';' Conditions ';' Math ')' '{'
               Lines '}'"
218
219     def p_Conditions1(p):
220         "Conditions : Condition"
221         parser.C = parser.C + 1
222         parser.aux.append("COND")
223
224     def p_Conditions2(p):
225         "Conditions : Condition AND Conditions"
226         c = parser.aux.pop()
227         b = parser.aux.pop()
228         a = parser.aux.pop()
229         s = a + b + "AND\n"
230         parser.aux.append(s)
231         parser.aux.append(c)
232
233     def p_Conditions3(p):
234         "Conditions : Condition OR Conditions"
235         c = parser.aux.pop()
236         b = parser.aux.pop()
237         a = parser.aux.pop()
238         s = a + b + "OR\n"
239         parser.aux.append(s)
240         parser.aux.append(c)
241
242     def p_Condition1(p):
243         "Condition : Expression EQ Expression"
244         b = parser.aux.pop()
245         a = parser.aux.pop()
246         s = a + b + "EQUAL\n"
247         parser.aux.append(s)
248
249     def p_Condition2(p):
250         "Condition : Expression NEQ Expression"
251         b = parser.aux.pop()
252         a = parser.aux.pop()
253         s = a + b + "EQUAL\nNOT\n"
254         parser.aux.append(s)
```

```
255
256     def p_Condition3(p):
257         "Condition : Expression LT Expression"
258         b = parser.aux.pop()
259         a = parser.aux.pop()
260         s = a + b + "INF\n"
261         parser.aux.append(s)
262
263     def p_Condition4(p):
264         "Condition : Expression LE Expression"
265         b = parser.aux.pop()
266         a = parser.aux.pop()
267         s = a + b + "INFEQ\n"
268         parser.aux.append(s)
269
270     def p_Condition5(p):
271         "Condition : Expression GT Expression"
272         b = parser.aux.pop()
273         a = parser.aux.pop()
274         s = a + b + "SUP\n"
275         parser.aux.append(s)
276
277     def p_Condition6(p):
278         "Condition : Expression GE Expression"
279         b = parser.aux.pop()
280         a = parser.aux.pop()
281         s = a + b + "SUPEQ\n"
282         parser.aux.append(s)
283
284     def p_Condition7(p):
285         "Condition : NOT '(' Condition ')'"
286         a = parser.aux.pop()
287         s = a + "NOT\n"
288         parser.aux.append(s)
289
290     def p_Math1(p):
291         "Math : Atribuition"
292
293     def p_Math2(p):
294         "Math : Atribuition ',' Math"
295
296     def p_Read(p):
297         "Read : READ '(' STRING ',' Address ')' ';'"
298         a = parser.aux.pop()
299         s = "READ\nATOI" + a
300         parser.aux.append(s)
301
302     def p_Address(p):
```

```
303         "Address : '&' ID"
304         s = f"STOREG {parser.reg.index(p[2])}\n"
305         parser.aux.append(s)
306 
307     def p_Write1(p):
308         "Write : WRITE '(' STRING ')' ';'"
309         s = f"PUSHS {p[3]}\nWRITES\n"
310         parser.aux.append(s)
311 
312     def p_Write2(p):
313         "Write : WRITE '(' STRING ',' Addresses ')' ';'"
314         a = p[3].split("%d")
315         s = f"PUSHS {a.pop()}\"\n"
316         for i in range(len(a)):
317             s = s + parser.aux.pop() + f"PUSHS {a.pop()}\"\nCONCAT\n"
318             i = i + 1
319         s = s + "WRITES\n"
320         parser.aux.append(s)
321 
322     def p_Addresses1(p):
323         "Addresses : ID"
324         s = f"PUSHG {parser.reg.index(p[1])}\nSTRI\nCONCAT\n"
325         parser.aux.append(s)
326 
327     def p_Addresses2(p):
328         "Addresses : ID ',' Addresses"
329         e = "PUSHS \" \"\nCONCAT\n"
330         s = e + f"PUSHG {parser.reg.index(p[1])}\nSTRI\nCONCAT\n"
331         parser.aux.append(s)
332 
333     def p_Output(p):
334         "Output : RETURN Ret ';'"
335         parser.type.pop()
336         r = parser.aux.pop()
337         parser.aux.pop()
338         parser.aux.append(r + "RETURN\n")
339 
340     def p_Ret1(p):
341         "Ret : Expression"
342 
343     def p_Ret2(p):
344         "Ret : "
345         pass
346 
347     def p_error(p):
348         if p:
349             print(f"ERRO SINTÁTICO :'{p.value}'\nReescreva a frase")
350         else:
```

```
351             print("ERRO SINTÁTICO: token inesperado")
352         parser.exito = False
353 
354     parser = yacc.yacc()
355     parser.exito = True
356     parser.c = parser.C = 0
357     parser.reg = []
358     parser.type =[]
359     parser.aux = []
360     parser.mv = ""
361 
362     fonte = ""
363     c = open("teste2.c", "r")
364     for linha in c:
365         fonte += linha
366     c.close()
367     parser.parse(fonte)
368 
369     with open("mv.txt", "w") as a:
370         a.write(parser.mv)
371 
372     if parser.exito:
373         print("Parsing terminou com sucesso.\nCompilação Concluída.")
```

5 Testes e Resultados

5.1 Testes

Com o *lexer* e o *Parser* prontos, procedemos para a fase de testes do nosso compilador. Para isso, escrevemos alguns ficheiros em C. Por exemplo:

```c
#include <stdio.h>

int main() {
    int a = 3;
    int b = 4;
    int m, M, r;
    int i = 0;
    if (a<b){
        m = a;
        M = b;
        r = b;
    }
    else{
        m = b;
        M = a;
        r = a;
    }
    while(i<m-1){
        r = r + M;
        i = i + 1;
    }
    printf("O resultado é: %d", r);
    return 0;
}
```

Ou então:

```c
#include <stdio.h>

int f(){
    return 3;
}

```

```
int main(){
    int a, b;
    printf("Val: ");
    scanf("%d", &a);
    b = f();
    if (a>b){
        b = a*b;
    }
    printf("A:%d B:%d\n", a, b);
    return 0;
}
```

Estes códigos, e outros similares, serviram para testar a correção do nosso compilador. Com os dois códigos combinados, temos declarações de variáveis, operações aritméticas e lógicas, leitura/escrita de dados, estruturas de controle de fluxo (ambos seleção e iteração) e chamada de funções sem parâmetros.

Dessa forma, testamos todos as diferentes comandos e possibilidades descritas no anteriormente.

5.2 Resultados

Para o primeiro código C, o resultado foi:

```
main:
PUSHI 3
PUSHI 4
PUSHI 0 //r
PUSHI 0 //M
PUSHI 0 //m
PUSHI 0
START
PUSHG 0
PUSHG 1
INF
JZ Else
PUSHG 0
STOREG 4
PUSHG 1
STOREG 3
PUSHG 1
STOREG 2
JUMP End1
Else: //NOP
PUSHG 1
STOREG 4
PUSHG 0
STOREG 3
```

```
    PUSHG 0
    STOREG 2
    End1: //NOP
    Flag: //NOP
    PUSHG 5
    PUSHG 4
    PUSHI 1
    SUB
    INF
    JZ End2:
    PUSHG 2
    PUSHG 3
    ADD
    STOREG 2
    PUSHG 5
    PUSHI 1
    ADD
    STOREG 5
    JUMP Flag
    End2: //NOP
    PUSHS ""
    PUSHG 2
    STRI
    CONCAT
    PUSHS "O resultado é: "
    CONCAT
    WRITES
    PUSHI 0
    STOP
```

Ao correr este código na Máquina Virtual (EWVM) averiguamos que, de facto, o resultado está certo. O resultado final de ambos os códigos é o mesmo.

Quanto ao segundo código:

```
    f:
    PUSHI 3
    RETURN
    main:
    PUSHI 0 //b
    PUSHI 0 //a
    START
    PUSHS "Val: "
    WRITES
    READ
    ATOISTOREG 1
    PUSHA f
    CALL
    STOREG 0
```

```
15      PUSHG 1
16      PUSHG 0
17      SUP
18      JZ Else
19      PUSHG 1
20      PUSHG 0
21      MUL
22      STOREG 0
23      JUMP End1
24      AUXPUSHS \n""
25      PUSHS " "
26      CONCAT
27      PUSHG 1
28      STRI
29      CONCAT
30      PUSHS  B:"
31      CONCAT
32      PUSHG 0
33      STRI
34      CONCAT
35      PUSHS "A:"
36      CONCAT
37      WRITES
38      PUSHI 0
39      STOP
```

Da mesma forma, é fácil de confirma que o resultado da compilação do segundo código C também está correto.

6 Conclusão

A concepção deste trabalho prático permitiu-nos combinar ideias e técnicas relacionadas com a tecnologia da linguagem na programação gramatical. Implementações de linguagem simples, com suporte a variáveis atômicas, instruções algorítmicas básicas, controle de fluxo e outros recursos adicionais, destacaram desafios e melhores práticas no desenvolvimento de gramáticas de tradução e no uso de ferramentas como Lex e Yacc do PLY.

O pseudocódigo de design e máquina virtual (EWVM) forneceu uma explicação prática da tradução, destacando a possibilidade de construir e transformar a gramática em uma solução eficaz. A utilização de técnicas e conceitos de processamento de texto não só nos permitiu compreender melhor os conceitos teóricos, mas também aplicá-los de forma eficaz na resolução de problemas práticos. É também definida uma linguagem(com base em C) que atenda aos requisitos do enunciado, incluindo manipulação e métodos básicos de controle. O documento LaTeX contribuiu para a transparência na apresentação do projeto, garantindo que os processos e resultados fossem bem documentados e reproduzíveis.

Por fim, o trabalho produzido revelou-se uma experiência de aprendizagem, afirmando as competências técnicas e práticas necessárias para projetos mais complexos.O projeto desenvolvido, com exemplos práticos, atingiu seus objetivos e apresentou avanços significativos na compreensão e aplicação dos conceitos lecionados nesta unidade curricular.