Relatório de Desenvolvimento

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

1970-01-01

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 e 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 de casos de teste.

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.

Contextualização

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.

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 os negócios. 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.

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'=='
    return t

def t_NEQ(t):
    r'\!='
    return t

def t_NOT(t):
    r'\!(?!=)'
    return t

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

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

def t_LT(t):
    r'<(?!=)'
    return t

def t_GT(t):
    r'>(?!=)'
    return t

def t_AND(t):
    r'&&'
    return t
```

```
def t_OR(t):
    r'\|\|'
    return t

def t_ATRIBUICAO(t):
    r'=(?!=)'
    return t

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

def t_INCLUDE(t):
    r'\#[ ]?include'
    return t

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

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

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

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

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

def t_WRITE(t):
    r'printf'
    return t

def t_READ(t):
    r'scanf'
    return t

def t_STRING(t):
    r'\".+\"'
    return t

def t_INT(t):
```

```
        r'(-)?[0-9]+(?!\.)'
        return t

    def t_ID(t):
        r'[A-z][A-z0-9_]*'
        return t

    t_ignore = ' \n\t'

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

    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.

Análisador Sintático

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
      | €
```

Máquina Virtual

De acordo com o enunciado do trabalho, foi nos pedido que fosse gerado um código assembly por isso foi necessario acrescentar certas especificações 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"

def p_Funcs1(p):
    "Funcs : Func"

def p_Funcs2(p):
    "Funcs : Func Funcs"

def p_Func(p):
    "Func : Tipo ID '(' ')' '{' Declarations Lines Output '}'"
    o = parser.aux.pop()
    l = parser.aux.pop()
    d = parser.aux.pop()
    f = f"{p[2]}:\n"
    if p[2] == "main":
        o = o.replace("RETURN", "STOP")
        f = f+d+l+o
    else:
        f = f+l+o
    #parser.aux.append(f)
    parser.mv = parser.mv + f
    parser.aux.clear()

def p_Tipo(p):
    "Tipo : INTT"
    parser.type.append("PUSHI")

def p_Declarations1(p):
    "Declarations : "
    s = ""
    for c in parser.aux:
```

```
            s = s + c
        s = s + "START\n"
        parser.aux = []
        parser.aux.append(s)
        parser.aux.append("AUX")
        pass

    def p_Declarations2(p):
        "Declarations : Declaration Declarations"

    def p_Declaration1(p):
        "Declaration : Tipo VarList ';'"
        parser.type.pop()

    def p_Declaration2(p):
        "Declaration : Tipo ID ATRIBUICAO Expression ';'"
        parser.type.pop()
        if p[2] not in parser.reg:
            parser.reg.append(p[2])
        else:
            parser.aux.append(f"ERR \"Variável {p[1]} já declarada\"\
n")

    def p_VarList1(p):
        "VarList : ID "
        if p[1] not in parser.reg:
            parser.reg.append(p[1])
            t = parser.type[-1]
            parser.aux.append(f"{t} 0 //{p[1]}\n")
        else:
            parser.aux.append(f"ERR \"Variável {p[1]} já declarada\"\
n")

    def p_VarList2(p):
        "VarList : ID ',' VarList"
        if p[1] not in parser.reg:
            parser.reg.append(p[1])
            t = parser.type[-1]
            parser.aux.append(f"{t} 0 //{p[1]}\n")
        else:
            parser.aux.append(f"ERR \"Variável {p[1]} já declarada\"\
n")

    def p_Expression1(p):
        "Expression : Expression ADD Expression"
        b = parser.aux.pop()
        a = parser.aux.pop()
        s = a + b + "ADD\n"
        parser.aux.append(s)
```

```
def p_Expression2(p):
    "Expression : Expression SUB Expression"
    b = parser.aux.pop()
    a = parser.aux.pop()
    s = a + b + "SUB\n"
    parser.aux.append(s)

def p_Expression3(p):
    "Expression : Expression MUL Expression"
    b = parser.aux.pop()
    a = parser.aux.pop()
    s = a + b + "MUL\n"
    parser.aux.append(s)

def p_Expression4(p):
    "Expression : Expression DIV Expression"
    b = parser.aux.pop()
    a = parser.aux.pop()
    s = a + b + "DIV\n"
    parser.aux.append(s)

def p_Expression5(p):
    "Expression : '(' Expression ')'"

def p_Expression6(p):
    "Expression : ID"
    if p[1] in parser.reg:
        s = f"PUSHG {parser.reg.index(p[1])}\n"
        parser.aux.append(s)
    else:
        parser.aux.append("ERR \"Var não declarada\"\n")

def p_Expression7(p):
    "Expression : Value"

def p_Expression8(p):
    "Expression : Call"

def p_Value1(p):
    "Value : INT"
    t = parser.type[-1]
    if t == "PUSHI":
        s = f"{t} {p[1]}\n"
        parser.aux.append(s)
    else:
        parser.aux.append("ERR \"Valor não é inteiro\"\n")

def p_Call(p):
    "Call : ID '(' ')'"
    s = f"PUSHA {p[1]}\nCALL\n"
```

```
    parser.aux.append(s)

def p_Lines1(p):
    "Lines : "
    s = ""
    c = parser.aux.pop()
    while c != "COND" and c != "AUX":
        s = c + s
        c = parser.aux.pop()
    parser.aux.append(s)
    parser.aux.append("AUX")
    pass

def p_Lines2(p):
    "Lines : Line Lines"

def p_Line1(p):
    "Line : Atribuition"

def p_Line2(p):
    "Line : Select"
    parser.c = parser.c + 1
    if parser.c == parser.C:
        parser.c = 0

def p_Line3(p):
    "Line : Cicle"
    parser.c = parser.c + 1
    if parser.c == parser.C:
        parser.c = 0

def p_Line4(p):
    "Line : Read"

def p_Line5(p):
    "Line : Write"

def p_Line6(p):
    "Line : COMENT"
    f"{p[1]}\n"

def p_Atribuition(p):
    "Atribuition : ID ATRIBUICAO Expression ';'"
    s = f"STOREG {parser.reg.index(p[1])}\n"
    parser.aux[-1] = parser.aux[-1] + s

def p_Select(p):
    "Select : IF '(' Conditions ')' '{' Lines '}' Else"
    e = parser.aux.pop()
    i = parser.aux.pop()
```

```
        c = parser.aux.pop()
        s = c + "JZ Else\n" + i + f"JUMP End{parser.C-parser.c}\n" + e
        parser.aux.append(s)

    def p_Else1(p):
        "Else : ELSE '{' Lines '}'"
        parser.aux.pop()
        e = parser.aux.pop()
        s = "Else: //NOP\n" + e + f"End{parser.C-parser.c}: //NOP\n"
        parser.aux.append(s)

    def p_Else2(p):
        "Else : "
        pass

    def p_Cicle1(p):
        "Cicle : WHILE '(' Conditions ')' '{' Lines '}'"
        parser.aux.pop()
        cc = parser.aux.pop()
        c = parser.aux.pop()
        s = "Flag: //NOP\n" + c + f"JZ End{parser.C-parser.c}:\n" + cc
+f"JUMP Flag\nEnd{parser.C-parser.c}: //NOP\n"
        parser.aux.append(s)

    def p_Cicle2(p):
        "Cicle : FOR '(' ID ATRIBUICAO INT ';' Conditions ';' Math ')'
'{' Lines '}'"

    def p_Conditions1(p):
        "Conditions : Condition"
        parser.C = parser.C + 1
        parser.aux.append("COND")

    def p_Conditions2(p):
        "Conditions : Condition AND Conditions"
        c = parser.aux.pop()
        b = parser.aux.pop()
        a = parser.aux.pop()
        s = a + b + "AND\n"
        parser.aux.append(s)
        parser.aux.append(c)

    def p_Conditions3(p):
        "Conditions : Condition OR Conditions"
        c = parser.aux.pop()
        b = parser.aux.pop()
        a = parser.aux.pop()
        s = a + b + "OR\n"
        parser.aux.append(s)
        parser.aux.append(c)
```

```
def p_Condition1(p):
    "Condition : Expression EQ Expression"
    b = parser.aux.pop()
    a = parser.aux.pop()
    s = a + b + "EQUAL\n"
    parser.aux.append(s)

def p_Condition2(p):
    "Condition : Expression NEQ Expression"
    b = parser.aux.pop()
    a = parser.aux.pop()
    s = a + b + "EQUAL\nNOT\n"
    parser.aux.append(s)

def p_Condition3(p):
    "Condition : Expression LT Expression"
    b = parser.aux.pop()
    a = parser.aux.pop()
    s = a + b + "INF\n"
    parser.aux.append(s)

def p_Condition4(p):
    "Condition : Expression LE Expression"
    b = parser.aux.pop()
    a = parser.aux.pop()
    s = a + b + "INFEQ\n"
    parser.aux.append(s)

def p_Condition5(p):
    "Condition : Expression GT Expression"
    b = parser.aux.pop()
    a = parser.aux.pop()
    s = a + b + "SUP\n"
    parser.aux.append(s)

def p_Condition6(p):
    "Condition : Expression GE Expression"
    b = parser.aux.pop()
    a = parser.aux.pop()
    s = a + b + "SUPEQ\n"
    parser.aux.append(s)

def p_Condition7(p):
    "Condition : NOT '(' Condition ')'"
    a = parser.aux.pop()
    s = a + "NOT\n"
    parser.aux.append(s)

def p_Math1(p):
```

```
    "Math : Atribuition"

def p_Math2(p):
    "Math : Atribuition ',' Math"

def p_Read(p):
    "Read : READ '(' STRING ',' Address ')' ';'"
    a = parser.aux.pop()
    s = "READ\nATOI" + a
    parser.aux.append(s)

def p_Address(p):
    "Address : '&' ID"
    s = f"STOREG {parser.reg.index(p[2])}\n"
    parser.aux.append(s)

def p_Write1(p):
    "Write : WRITE '(' STRING ')' ';'"
    s = f"PUSHS {p[3]}\nWRITES\n"
    parser.aux.append(s)

def p_Write2(p):
    "Write : WRITE '(' STRING ',' Addresses ')' ';'"
    a = p[3].split("%d")
    s = f"PUSHS {a.pop()}\"\n"
    for i in range(len(a)):
        s = s + parser.aux.pop() + f"PUSHS {a.pop()}\"\nCONCAT\n"
        i = i + 1
    s = s + "WRITES\n"
    parser.aux.append(s)

def p_Addresses1(p):
    "Addresses : ID"
    s = f"PUSHG {parser.reg.index(p[1])}\nSTRI\nCONCAT\n"
    parser.aux.append(s)

def p_Addresses2(p):
    "Addresses : ID ',' Addresses"
    e = "PUSHS \" \"\nCONCAT\n"
    s = e + f"PUSHG {parser.reg.index(p[1])}\nSTRI\nCONCAT\n"
    parser.aux.append(s)

def p_Output(p):
    "Output : RETURN Ret ';'"
    parser.type.pop()
    r = parser.aux.pop()
    parser.aux.pop()
    parser.aux.append(r + "RETURN\n")

def p_Ret1(p):
```

```
    "Ret : Expression"

def p_Ret2(p):
    "Ret : "
    pass

def p_error(p):
    if p:
        print(f"ERRO SINTÁTICO :'{p.value}'\nReescreva a frase")
    else:
        print("ERRO SINTÁTICO: token inesperado")
    parser.exito = False

parser = yacc.yacc()
parser.exito = True
parser.c = parser.C = 0
parser.reg = []
parser.type =[]
parser.aux = []
parser.mv = ""

fonte = ""
c = open("teste2.c", "r")
for linha in c:
    fonte += linha
c.close()
parser.parse(fonte)

with open("mv.txt", "w") as a:
    a.write(parser.mv)

if parser.exito:
    print("Parsing terminou com sucesso.\nCompilação Concluída.")
```

Testes e Resultados

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:

```
#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:

```
#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.

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
PUSHG 1
PUSHG 0
SUP
JZ Else
PUSHG 1
PUSHG 0
MUL
STOREG 0
JUMP End1
AUXPUSHS \n""
PUSHS " "
CONCAT
PUSHG 1
STRI
CONCAT
PUSHS  B:"
CONCAT
PUSHG 0
STRI
CONCAT
PUSHS "A:"
CONCAT
WRITES
PUSHI 0
STOP
```

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

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 definida uma linguagem(com base em C) que atenda aos requisitos, incluindo manipulação e métodos básicos de controle. Os documentos LaTeX têm contribuído para a transparência na apresentação dos projetos técnicos, garantindo que os processos e resultados sejam 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. A trabalho desenvolvido, com exemplos práticos, atingiu seus objetivos e apresentou avanços significativos na compreensão e aplicação dos conceitos lecionados nesta unidade curricular.