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

Trabalho *Prático 2*

Relatório de Desenvolvimento

Grupo 09

Eduardo Pereira	Diogo Coelho	Tomás Meireles
(A70619)	(A100092)	(A100106)

12 de janeiro de 2025

Resumo

Este relatório descreve o desenvolvimento de um projeto, no âmbito da Unidade Curricular de Processamento de Linguagens e Compiladores, do 3º ano da Licenciatura em Ciências da Computação, na Universidade do Minho.

Este projeto consiste em duas grandes componentes: a criação de uma Linguagem de Programação e a implementação de um compilador capaz da compreensão da mesma.

O compilador é responsável pela geração do código assembly relativo a cada função compilada, código este que pode ser corrido numa máquina virtual de forma a verificar o output das funções. Foram utilizados os módulos Lex e Yacc do Python para a realização deste projeto.

Conteúdo

Capítulo 1

Introdução

No âmbito da Unidade Curricular de Processamento de Linguagens e Compiladores, foi-nos proposto, pelo docente, a implementação de um compilador capaz de reconhecer e interpretar programas escritos escritos numa linguagem por nós criada, uma linguagem simples, imperativa.

O compilador gera código assembly a partir dos programas interpretados, código este que preserva a semântica do programa e que pode ser inserido numa máquina virtual para execução (**Máquina Virtual VM**).

1.1 Objetivo

O presente documento tem como objetivo servir de guia para o leitor, isto é, para demonstrar como poderia ser escrito um programa nesta linguagem, de forma a ser corretamente interpretado pelo compilador desenvolvido.

1.2 Estrutura do Relatório

De forma a facilitar a leitura e compreensão deste documento, nesta secção será explicada a sua estrutura e o conteúdo de cada um dos capítulos resumido e explicado.

- **Capítulo 1** - Definição do Sistema, que consiste em: Introduzir e descrever a contextualização do projeto desenvolvido, assim como os objetivos a atingir com o mesmo.
- **Capítulo 2** - Análise e Especificação: Descrever informalmente o problema, indicando o que se espera ser possível fazer.
- **Capítulo 3** - Concepção/Desenho da Resolução: Expôr todas as estruturas de dados utilizadas, que, de uma forma prática, indicam que tipo de programas podem (ou não) ser criados nesta linguagem.
- **Capítulo 4** - Desenho da Gramática: A gramática consiste numa demonstração mais organizada do que o compilador desenvolvido aceita.
- **Capítulo 5** - O Sistema Desenvolvido: Neste capítulo é apresentado o código de implementação do projeto, dividido por categorias, assim como uma breve explicação das suas funções.
- **Capítulo 6** - Codificação e Testes: Exemplificação de código reconhecido pelo nosso interpretador, e o seu respetivo código em Assembly.
- **Capítulo 7** - Conclusão: Contém as últimas impressões acerca do projeto desenvolvido e oportunidades de trabalho futuro.
- **Apêndice A** - Código Lex: É apresentado o código inteiro do analisador léxico.
- **Apêndice B** - Código Yacc: É apresentado o código, agora como um todo, do Código Parser e Tradutor.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

Pretende-se definir uma linguagem de programação imperativa, espera-se que esta seja capaz de:

- Declarar variáveis atómicas do tipo inteiro;
- Efetuar operações aritméticas, relacionais e lógicas;
- Efetuar instruções algorítmicas, por exemplo, efetuar atribuições a variáveis;
- Efetuar operações de leitura do *standard input* e escrita no *standard output*;
- Efetuar instruções de seleção para controlo de fluxo de execução, por exemplo, condições *if-then-otherwise*;
- Efetuar instruções de repetição para controlo de fluxo de execução, por exemplo, ciclos *while-do*;
- Declarar e alterar variáveis do tipo array de inteiros, em relação aos quais seja apenas possível a operação de indexação;

Pretende-se, também, desenvolver um compilador para a linguagem criada, com base na **GIC** criada, com recurso aos módulos **Lex/Yacc** do **PLY/Python**. Este compilador deve gerar *Assembly* da **Máquina Virtual VM**.

Capítulo 3

Concepção/Desenho da Resolução

3.1 Estruturas de Dados

3.1.1 Declarações

Expressão	**Significado**
`var x`	Declara variável x do tipo inteiro
`var arr[10]`	Declara variável arr do tipo array de tamanho 10
`var mat[5][5]`	Declara variável mat do tipo matriz com 5 linhas e 5 colunas

3.1.2 Atribuições

Expressão	**Significado**
`x = 10`	Atribui o valor 10 à variável x
`arr[5] = 3`	Atribui o valor 3 à posição 5 do array arr
`mat[5][5] = 3`	Atribui o valor 3 à posição (5,5) da matriz mat

3.1.3 Operações

Aritméticas

Expressão	Significado
`x + y`	Soma x com y
`x - y`	Subtrai y a x
`x * y`	Multiplica x por y
`x / y`	Divisão inteira de x por y
`x % y`	Resto da divisão inteira de x por y

Relacionais

Expressão	Significado
`x > y`	Calcula x maior que y
`x >= y`	Calcula x maior ou igual a y
`x < y`	Calcula x menor que y
`x <= y`	Calcula x menor ou igual a y
`x == y`	Calcula x igual a y
`x != y`	Calcula x diferente de y

Lógicas

Expressão	Significado
`x && y`	Calcula a conjunção de x e y ($x \wedge y$)
`x || y`	Calcula a disjunção de x e y ($x \vee y$)

3.1.4 Controlo de fluxo de execução: Instruções de seleção

If-Then:

```
if (Condicao) then {Corpo}
```

Se **Condicao** tiver valor verdade, executa **Corpo**, caso contrário, este não é executado.

If-Then-Otherwise:

```
if (Condicao) then {Corpo1}
              otherwise {Corpo2}
```

Se **Condicao** tiver valor verdade, executa **Corpo1**, caso contrário, executa **Corpo2**. Em ambas as instruções de seleção, o corpo pode ter uma ou mais instruções.

3.1.5 Controlo de fluxo de execução: Instruções cíclicas

While-Do:

```
while (Condicao) do {Corpo}
```

Enquanto **Condicao** se verificar, isto é, tiver valor verdade, executa **Corpo**. Quando **Condicao** não se verificar, o ciclo acaba e **Corpo** já não é executado.

Repeat-Until:

```
repeat {Corpo} until (Condicao)
```

Esta instrução é o contrário do ciclo While-Do, ou seja, **Corpo** é repetido enquanto **Condicao** não se verificar. Quando esta se verificar, o ciclo acaba e **Corpo** já não é executado.

3.1.6 Acesso a variáveis

Inteiros:

```
x
```

Acesso a uma variável existente x.

Arrays (Índices):

```
arr[i]
```

Acesso ao valor no índice i do array arr.

Matrizes (Linhas, Colunas):

```
mat[i][j]
```

Acesso ao valor na linha i, coluna j, da matriz mat.

3.1.7 Input-Output

Expressão	**Significado**
`print x (Int)`	Imprime o inteiro x no *Standard Output*
`print arr (Array)`	Imprime o array arr no *Standard Output*
`print mat (Matriz)`	Imprime a matriz mat no *Standard Output*
`x=input`	Lê do *Standard Input* e coloca o valor na variável x
`arr[i]=input`	Lê do *Standard Input* e coloca o valor na posição i do array arr
`mat[i][j]=input`	Lê do *Standard Input* e coloca o valor na posição (i,j) da matriz mat

Capítulo 4

Desenho da Gramática

```
S : Program

Program : Header Code
        | Code

Header : Header Decl
       | Decl

Decl : VAR NameList ';'
     | VAR NAME '[' NUM ']' ';'
     | VAR NAME '[' Expr ']' '[' Expr ']' ';'

NameList : NAME
         | NameList ',' NAME


Code : Code Codes
     | Codes

Codes : Conditions
      | WhileDo
      | RepeatUntil
      | Assign
      | Print

Conditions : IF '(' Condition ')' THEN '{' Code '}'
           | IF '(' Condition ')' THEN '{' Code '}' OTHERWISE '{' Code '}'

WhileDo : WHILE '(' Condition ')' DO '{' Code '}'

RepeatUntil : REPEAT '{' Code '}' UNTIL '(' Condition ')' ';'

Assign : NAME '=' Expr ';'
       | NAME '[' Expr ']' '=' Expr ';'
       | NAME '[' Expr ']' '[' Expr ']' '=' Expr ';'
       | NAME '[' Expr ']' '=' INPUT ';'
       | NAME '[' Expr ']' '[' Expr ']' '=' INPUT ';'
       | NAME '=' INPUT ';'
```

```
Expr : Condition
     | Variable
     | NUM
     | Expr '+' Expr
     | Expr '-' Expr
     | Expr '*' Expr
     | Expr '/' Expr
     | Expr '%' Expr
     | '(' Expr ')'

Condition : Expr EQ Expr
          | Expr NEQ Expr
          | '!' Expr
          | Expr '>' Expr
          | Expr MOREEQ Expr
          | Expr '<' Expr
          | Expr LESSEQ Expr
          | Expr AND Expr
          | Expr OR Expr
          | '(' Condition ')'

Variable : NAME
         | NAME '[' Expr ']'
         | NAME '[' Expr ']' '[' Expr ']'

Print : PRINT NAME ';'
      | PRINT STRING ';'
```

Capítulo 5

O Sistema Desenvolvido

```
def p_Program(p):
    "Program : Header Code"
    p[0] = p[1] + "START\n" + p[2] + "STOP\n"
```

O Programa é constituído por **Cabeçalho** e **Corpo**.

```
def p_WOHeader(p):
    "Program : Code"
    p[0] = "START\n" + p[1] + "STOP\n"

def p_MultHeader(p):
    "Header : Header Decl"
    p[0] = p[1] + p[2]

def p_SingleHeader(p):
    "Header : Decl"
    p[0] = p[1]
```

O Cabeçalho pode conter um número indeterminado de declarações de variáveis, assim como pode não ter nenhuma, ou seja, só ter o Corpo do código.
Da mesma forma, um cabeçalho pode ter uma ou mais declarações.
O código Assembly tem sempre uma delimitação onde começa o corpo do programa, e onde acaba o programa, através de **START** e **STOP**.

```
def p_MultCode(p):
    "Code : Code Codes"
    p[0] = p[1] + p[2]

def p_SingleCode(p):
    "Code : Codes"
    p[0] = p[1]

def p_Codes(p):
    """Codes : Conditions
             | WhileDo
             | RepeatUntil
             | Assign
             | Print
    """
    p[0] = p[1]
```

Relativamente ao Corpo do programa, o Corpo pode ser constituído por apenas um bloco, ou por mais do que um. Cada bloco de código pode ser uma Condição, um ciclo **While-Do**, um ciclo **Repeat-Until**, uma atribuição, ou uma escrita para o *Standard Output*.
Estão também incluídas nas atribuições, as leituras do *Standard Input*.

5.1 Declarações

Variáveis inteiras

```
def p_IntDecl(p):
    "Decl : VAR NameList ';'"
    for name in p[2]:
        if name not in p.parser.trackmap:
            p.parser.trackmap.update({name: p.parser.memPointer})
            p[0] = (p[0] or "") + "PUSHI 0\n"#
            p.parser.memPointer += 1
        else:
            raise Exception(f"Variable {name} already declared.")

def p_NameList(p):
    """NameList : NAME
                | NameList ',' NAME"""
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[3]]
```

Podemos ter a declaração de apenas um inteiro de cada vez, ou uma `Namelist`, que permite declarar variáveis ao mesmo tempo, adicionando a uma lista de variáveis.
Para declarar um inteiro, verificamos se o seu **name**, isto é, cada variável em `NameList` (p[2]) já foi declarada. Caso tal não se verifique, adicionamos o seu **name** ao dicionário das variáveis, associando-lhe a sua posição na stack (`parser.memPointer`). Por fim, incrementamos o `parser.memPointer` com o número de células da stack necessárias para guardar um inteiro, neste caso, uma.
Se, pelo contrário, a variável já se encontrar declarada, então imprime o erro **"Variable name already declared."**, para informação do utilizador.

Arrays e Matrizes

```
def p_ArrayDecl(p):
    "Decl : VAR NAME '[' NUM ']' ';'"
    if p[2] not in p.parser.trackmap:
        p.parser.trackmap.update({p[2]: (p.parser.memPointer, int(p[4]))})
        memSpace = int(p[4])
        p[0] = f"PUSHN {memSpace}\n"
        p.parser.memPointer += int(p[4])
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[2]} is already declared.")
```

```
def p_MatrixDecl(p):
    "Decl : VAR NAME '[' NUM ']' '[' NUM ']' ';'"
    if p[2] not in p.parser.trackmap:
        p.parser.trackmap.update({p[2]: (p.parser.memPointer, int(p[4]), int(p[7]))})
        memSpace = int(p[4]) * int(p[7])
        p[0] = f"PUSHN {memSpace}\n"
        p.parser.memPointer += memSpace
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[2]} is already declared.")
```

Para declarar um array ou uma matriz, verificamos se o seu **name** já foi declarado. Caso tal não se verifique, adicionamos o seu **name** ao dicionário das variáveis, associando-lhe a sua posição na stack (`parser.memPointer`). Por fim, incrementamos o `parser.memPointer` com o número de células da stack necessárias para guardar a estrutura, em ambos os casos, incrementa-se com a variável **memSpace**.
No caso de Array, é incrementado com **int(p[4])** e numa Matriz, com **int(p[4])*int(p[7])**.
Mais uma vez, se a variável já se encontrar declarada, então imprime o erro **"name is already declared."**, para informação do utilizador.

5.2 Atribuições

Todo o código de atribuição conta com controlo de declarações, isto é, antes de fazer qualquer tipo de atribuição, é testado se a variável já existe no dicionário parser.trackMap. É também testado se o tipo das variáveis corresponde ao que queremos atribuir.
O código de atribuição também conta com controlo de erros, imprimindo códigos de erro consoante a situação, de forma a informar o utilizador.

Variáveis inteiras

```
def p_ExpressionAssign(p):
    "Assign : NAME '=' Expr ';'"
    if p[1] in p.parser.trackmap:
        var = p.parser.trackmap.get(p[1])
        if type(var) == int:
            p[0] = p[3] + f"STOREG {var}\n"
        else:
            raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not an integer.")
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")
```

Armazena-se o valor no endereço da memória, através de **'STOREG{var}'**.

Arrays

```
def p_ArrayAssign(p):
     "Assign : NAME '[' Expr ']' '=' Expr ';'"
     if p[1] in p.parser.trackmap:
          varInfo = p.parser.trackmap.get(p[1])
          if len(varInfo) == 2:
                p[0] = f'PUSHGP\nPUSHI {varInfo[0]}\nPADD\n' + p[3] + p[6] + 'STOREN\
                   n'
          else:
               raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not an array.")
     else:
          raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")
```

Este código implementa a construção da atribuição a elementos de um array. Se a variável é um array válido, o endereço base do array é obtido através de `'PUSHGP'` seguido de `'PUSHI'` com o deslocamento `'varInfo[0]'`. `'PADD'` calcula o endereço base do array. O índice do array é avaliado e adicionado ao cálculo do endereço. O valor a ser atribuído (`p[6]`) é armazenado no endereço calculado com o comando `STOREN`.

Matrizes

```
def p_MatrixAssign(p):
     "Assign : NAME '[' Expr ']' '[' Expr ']' '=' Expr ';'"
     if p[1] in p.parser.trackmap:
            varInfo = p.parser.trackmap.get(p[1])
            if len(varInfo) == 3:
                  p[0] = f'PUSHGP\nPUSHI {varInfo[0]}\nPADD\n{p[3]}PUSHI {varInfo
                      [2]}\nMUL\n{p[6]}ADD\n{p[9]}STOREN\n'
            else:
                  raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not a matrix.")
     else:
          raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared."))
```

Este código implementa a atribuição a elementos de uma matriz, lidando com expressões indexadas na forma `Nome[Expr][Expr] = Expr`.
O endereço base da matriz obtém-se a partir de `'PUSHGP'` somado ao deslocamento `'varInfo[0]'` com `'PUSHI'`. O índice da linha (`p[3]`) é multiplicado pelo número de colunas (`varInfo[2]`) usando o comando `'MUL'`. O índice da coluna (`p[6]`) é somado ao resultado da multiplicação com `'ADD'`. O valor final de `p[9]` é armazenado no endereço calculado utilizando o comando `'STOREN'`.

5.3 Expressões

```
def p_Expr_condition(p):
    'Expr : Condition'
    p[0] = p[1]

def p_Expr_Variable(p):
    'Expr : Variable'
    p[0] = p[1]

def p_expression_number(p):
    "Expr : NUM"
    p[0] = f"PUSHI {p[1]}\n"

def p_Expr_base(p):
    "Expr : '(' Expr ')'"
    p[0] = p[2]
```

Cada expressão pode ser: Uma condição, variável, número inteiro ou simplesmente uma expressão entre parêntesis. No caso de ser um inteiro, gera-se o código Assembly `'PUSHI {p[1]}'`.

Pode também ser uma ou mais operações entre Expressões, que é explicado na secção a seguir.

5.4 Operações

Aritméticas

```
def p_Expr_OP(p):
    """Expr : Expr "+" Expr
            | Expr "-" Expr
            | Expr "*" Expr
            | Expr "/" Expr
            | Expr "%" Expr
    """
    if (p[2] == '+'):
        p[0] = p[1] + p[3] + "ADD \n"
    elif (p[2] == '-'):
        p[0] = p[1] + p[3] + "SUB \n"
    elif (p[2] == '*'):
        p[0] = p[1] + p[3] + "MUL \n"
    elif (p[2] == '/'):
        p[0] = p[1] + p[3] + "DIV \n"
    elif (p[2] == '%'):
        p[0] = p[1] + p[3] + "MOD \n"
```

O código acima concatena as expressões (p[1] e p[3]) e gera o Assembly consoante a operação aritmética explicitada.

Lógicas e Relacionais

```
def p_condLog(p):
    """Condition : Expr EQ Expr
                 | Expr NEQ Expr
                 | '!' Expr
                 | Expr '>' Expr
                 | Expr MOREEQ Expr
                 | Expr '<' Expr
                 | Expr LESSEQ Expr
                 | Expr AND Expr
                 | Expr OR Expr
                 """

    if (p[2] == "=="):
        p[0] = p[1] + p[3] + "EQUAL \n"
    elif (p[2] == "!="):
          p[0] = p[1] + p[3] + "EQUAL\nNOT \n"
    elif (p[1] == '!'):
        p[0] = p[2] + "NOT \n"
    elif (p[2] == '>'):
        p[0] = p[1] + p[3] + "SUP \n"
    elif (p[2] == ">="):
        p[0] = p[1] + p[3] + "SUPEQ \n"
    elif (p[2] == '<'):
        p[0] = p[1] + p[3] + "INF \n"
    elif (p[2] == "<="):
        p[0] = p[1] + p[3] + "INFEQ \n"
    elif (p[2] == "&&"):
        p[0] = p[1] + p[3] + 'ADD\nPUSHI 2\nEQUAL\n'
    elif (p[2] == "||"):
        p[0] = p[1] + p[3] + "ADD\nPUSHI 1\nSUPEQ\n"
    elif (p[1] == "Var"):
        p[0] = p[1]
```

O código acima concatena as expressões (p[1] e p[3]) e gera o Assembly consoante a operação explicitada, com a exceção de `'!Expr'`, que apenas concatena a expressão p[2] com `'NOT \n'`.

5.5 Instruções de seleção

Estas são as **Conditions** do nosso projeto, dividem-se em **If-Then** e **If-Then-Otherwise**.

```
def p_CondIfThen(p):
    "Conditions : IF '(' Condition ')' THEN '{' Code '}'"
    p[0] = p[3] + f"JZ l{p.parser.idLabel}\n" + p[7] + f"l{p.parser.idLabel}: NOP\n"
    p.parser.idLabel += 1

def p_CondIfThenOtherwise(p):
    "Conditions : IF '(' Condition ')' THEN '{' Code '}' OTHERWISE '{' Code '}'"
    p[0] = p[3] + f"JZ l{p.parser.idLabel}\n" + p[7] + f"JUMP l{p.parser.idLabel}f\nl
       {p.parser.idLabel}: NOP\n" + p[11] + f"l{p.parser.idLabel}f: NOP\n"
    p.parser.idLabel += 1
```

Depois de analisar a condição lógica p[3], caso esta tenha valor zero, é efetuado um salto para a *label* correspondente, através de **parser.idLabel**. Caso o valor não seja zero, o corpo p[7] é executado.
No caso do **otherwise**, existe mais uma instrução de salto, que é efetuado caso a condição p[3] tenha valor zero.
O **parse.idLabel** é sempre incrementado, o que é essencial para o salto ser efetuado para a parte correta do código.

5.6 Instruções cíclicas

Estas instruções dividem-se em ciclos **While-Do** e **Repeat-Until**.

```
def p_WhileDo(p):
    "WhileDo : WHILE '(' Condition ')' DO '{' Code '}'"
    p[0] = f"l{p.parser.idLabel}c: NOP\n" + p[3] + f"JZ l{p.parser.idLabel}f\n" + p
        [7] + f"JUMP l{p.parser.idLabel}c\nl{p.parser.idLabel}f: NOP\n"
    p.parser.idLabel += 1
```

Este código representa a construção da estrutura de repetição **WhileDo**. O ciclo é executado enquanto a `Condition` for verdadeira.
O label `l{p.parser.idLabel}c` marca o início da avaliação da condição, e a instrução **JZ** realiza um salto para `l{p.parser.idLabel}f` caso a condição não seja satisfeita, encerrando o ciclo. Se a condição for verdadeira, o bloco de código `p[7]` é executado, seguido por um salto de retorno ao início do ciclo com **JUMP**.
O incremento de `p.parser.idLabel` assegura que cada label seja único, permitindo uma correta execução e fluxo no código gerado.

```
def p_RepeatUntil(p):
    "RepeatUntil : REPEAT '{' Code '}' UNTIL '(' Condition ')' ';' "
    p[0] = ( f"l{p.parser.idLabel}c: NOP\n" + p[3]+ p[7]+ f"JZ l{p.parser.idLabel}c\n
        "+ f"JUMP l{p.parser.idLabel}f\n"+ f"l{p.parser.idLabel}f: NOP\n")
    p.parser.idLabel += 1
```

Este código implementa a lógica da estrutura de repetição **RepeatUntil**, onde o bloco de instruções `{Code}` é sempre executado antes da avaliação da condição `(Condition)`.
O label `l{p.parser.idLabel}c` marca o início do ciclo, e o comando **JZ** realiza um salto condicional para `l{p.parser.idLabel}f` caso a condição seja satisfeita. Se a condição não for verdadeira, o bloco `p[3]` é executado novamente, retornando ao início do ciclo através de **JUMP**.
A constante `p.parser.idLabel` é incrementada para garantir que cada label seja único e correto no fluxo de execução gerado.

5.7 Acesso a variáveis

```
def p_VarNum(p):
    "Variable : NAME"
    if p[1] in p.parser.trackmap:
        var = p.parser.trackmap.get(p[1])
        if type(var) == int:
            p[0] = f"PUSHG {var}\n"
        else:
            raise TypeError(f"Line {p.lineno(1)}, {p[1]} is not an integer.")
    else:
        raise Exception(f"Line {p.lineno(1)}, {p[1]} is not declared.")
```

Este código implementa a verificação e a manipulação de uma variável em um analisador sintático. Ao identificar um identificador `NAME`, ele verifica se o mesmo está declarado no mapa de variáveis (`trackmap`). Caso a variável exista e seja do tipo inteiro, o comando `PUSHG` é gerado, que carrega o valor global associado ao identificador.

Se a variável não for um inteiro, é levantada uma exceção de tipo (**TypeError**), indicando o erro de incompatibilidade no uso da variável. Caso a variável não esteja declarada, uma exceção genérica (**Exception**) é lançada, apontando que a mesma não foi encontrada no escopo.

Dessa forma, o código garante segurança na manipulação de variáveis e integridade no fluxo de execução gerado.

```
def p_VarArray(p):
    "Variable : NAME '[' Expr ']'"
    if p[1] in p.parser.trackmap:
        varInfo = p.parser.trackmap.get(p[1])
        if len(varInfo) == 2:
            p[0] = f'PUSHGP\nPUSHI {varInfo[0]}\nPADD\n' + p[3] + 'LOADN\n'
        else:
            raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not an array.")
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")
```

Este código implementa a construção do acesso a um elemento de um vetor (**Array**). Ao identificar a produção `Variable : NAME '[' Expr ']'`, o analisador verifica se o identificador `NAME` está declarado no mapa de variáveis (`trackmap`).

Caso a variável exista e seja um vetor (identificado pelo comprimento de `varInfo` igual a 2), o código gerado calcula o endereço do elemento do vetor com as seguintes instruções:

- `PUSHGP`: Empilha o ponteiro base do segmento global na pilha;
- `PUSHI`: Empilha o deslocamento inicial associado ao vetor;
- `PADD`: Soma o ponteiro base com o deslocamento para determinar o endereço inicial do vetor;
- `Expr`: Avalia o índice de acesso ao vetor;
- `LOADN`: Carrega o valor armazenado no endereço calculado.

Se a variável não for um vetor, é levantada uma exceção de tipo (`TypeError`), indicando que o uso é incompatível. Caso a variável não esteja declarada, uma exceção genérica é lançada, informando que o identificador não foi encontrado no escopo atual.

Dessa forma, o código assegura a correção no acesso aos elementos de vetores, tanto na validação semântica quanto na geração do fluxo de execução correspondente.

```
def p_VarMatrix(p):
    "Variable : NAME '[' Expr ']' '[' Expr ']'"
    if p[1] in p.parser.trackmap:
        varInfo = p.parser.trackmap.get(p[1])
        if len(varInfo) == 3:
            p[0] = f'PUSHGP\nPUSHI {varInfo[0]}\nPADD\n' + p[3] + f'PUSHI {varInfo
                [2]}\nMUL\n' + p[6] + 'ADD\n' + 'LOADN\n'
        else:
            raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not a matrix.")
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")
```

Esta função trata o acesso a elementos de uma matriz (**Matrix**) no analisador sintático. Quando a variável `NAME` é acessada com dois índices (`Expr` e `Expr`), o código verifica se a variável é declarada e possui estrutura de matriz (indicada por `len(varInfo) == 3`). O cálculo do endereço do elemento é feito através das instruções:

- `PUSHGP`: Carrega o ponteiro base do segmento global;
- `PUSHI`: Adiciona o deslocamento inicial da matriz;
- `PADD`: Calcula o início da matriz no espaço global;
- `MUL`: Calcula o deslocamento da linha, utilizando o número de colunas (`varInfo[2]`);
- `ADD`: Soma o deslocamento da linha ao índice da coluna;
- `LOADN`: Carrega o valor armazenado na posição calculada.

Caso a variável não seja uma matriz ou não esteja declarada, as exceções apropriadas são lançadas, garantindo que apenas estruturas válidas sejam manipuladas no código gerado.

5.8 Input-Output

Input

```
def p_Input_Array(p):
    "Assign : NAME '[' Expr ']' '=' INPUT ';'"
    if p[1] in p.parser.trackmap:
        var = p.parser.trackmap.get(p[1])
        if len(var) == 2:
            p[0] = f'PUSHGP\nPUSHI {var[0]}\nPADD\n' + p[3] + f'READ\nATOI\nSTOREN\n'
        else:
            raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not an array.")
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")
```

Esta função implementa a atribuição de entrada (**Input**) para um elemento de um vetor (**Array**). Ao identificar a produção `Assign : NAME '[' Expr ']' '=' INPUT ';'`, o analisador verifica se o identificador `NAME` é um vetor válido no mapa de variáveis (`trackmap`).
Se a variável for válida e do tipo vetor (`len(var) == 2`), o código gerado realiza as seguintes operações:

- Calcula o endereço do elemento do vetor utilizando `PUSHGP`, `PUSHI`, e `PADD`;
- Lê o valor de entrada com `READ`;
- Converte o valor lido de string para inteiro com `ATOI`;
- Armazena o valor no endereço calculado usando `STOREN`.

Se a variável não for um vetor ou não estiver declarada, exceções são lançadas, garantindo a validação semântica.

```
def p_Input_Matrix(p):
    "Assign : NAME '[' Expr ']' '[' Expr ']' '=' INPUT ';'"
    if p[1] in p.parser.trackmap:
        var = p.parser.trackmap.get(p[1])
        if len(var) == 3:
            p[0] = f'PUSHGP\nPUSHI {var[0]}\nPADD\n{p[3]}PUSHI {var[2]}\nMUL\n{p[6]}
                ADD\nREAD\nATOI\nSTOREN\n'
        else:
            raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not a matrix.")
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")
```

Esta função trata a atribuição de entrada (**Input**) para um elemento de uma matriz (**Matrix**). Ao identificar a produção `Assign : NAME '[' Expr ']' '[' Expr ']' '=' INPUT ';'`, o analisador verifica se o identificador `NAME` é uma matriz válida no mapa de variáveis (`trackmap`).
Caso a variável seja uma matriz (`len(var) == 3`), o código gerado realiza:

- Cálculo do endereço do elemento com `PUSHGP`, `PUSHI`, e `PADD`;
- Multiplicação do índice da linha pelo número de colunas (`MUL`);
- Soma do deslocamento da linha e coluna (`ADD`);
- Leitura do valor de entrada (`READ`);
- Conversão do valor para inteiro (`ATOI`);
- Armazenamento do valor na posição calculada (`STOREN`).

Se a variável não for uma matriz ou não estiver declarada, são levantadas exceções adequadas.

```
def p_Input_Var(p):
    "Assign : NAME '=' INPUT ';'"
    if p[1] in p.parser.trackmap:
        var = p.parser.trackmap.get(p[1])
        if type(var) == int:
            p[0] = f'READ\nATOI\nSTOREG {var}\n'
        else:
            raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not an integer.")
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")
```

Esta função implementa a atribuição de entrada (**Input**) para uma variável simples (**Var**). Ao identificar a produção `Assign : NAME '=' INPUT ';'`, o analisador verifica se o identificador `NAME` é uma variável declarada no mapa de variáveis (`trackmap`). Se a variável for válida e do tipo inteiro (`type(var) == int`), o código gerado realiza:

- Leitura do valor de entrada (`READ`);
- Conversão do valor de string para inteiro (`ATOI`);
- Armazenamento do valor na posição global associada (`STOREG`).

Output

```
def p_Print(p):
    "Print : PRINT NAME ';'"
    if p[2] in p.parser.trackmap:
        var = p.parser.trackmap.get(p[2])

        if type(var) == tuple:
            if len(var) == 2:
                array = f'PUSHS "[ "\nWRITES\n'
                for i in range(var[1]):
                    array += f'PUSHGP\nPUSHI {var[0]}\nPADD\nPUSHI {i}\nLOADN\nWRITEI
                        \nPUSHS " "\nWRITES\n'
                array += f'PUSHS "]"\nWRITES\n'
                p[0] = array + 'PUSHS "\\n"\nWRITES\n'

            elif len(var) == 3:
                matrix = ""
                for i in range(var[1]):
                    matrix += f'PUSHS "[ "\nWRITES\n'
                    for j in range(var[2]):
                        matrix += f'PUSHGP\nPUSHI {var[0]}\nPADD\nPUSHI {var[2] * i +
                            j}\nLOADN\nWRITEI\nPUSHS " "\nWRITES\n'
                    matrix += 'PUSHS "]\\n"\nWRITES\n'
                p[0] = matrix
        else:
            p[0] = f'PUSHG {var}\nWRITEI\nPUSHS "\\n"\nWRITES\n'
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[2]} is not declared.")
```

Esta função implementa a geração de código para a operação de saída (**Output**) com a produção `Print : PRINT NAME ';'`. Dependendo do tipo de variável `NAME`, diferentes estratégias são utilizadas:

Variável Simples: Se `NAME` for uma variável simples (inteiro), o código gerado realiza:- `PUSHG`: Empilha o valor global da variável;

- `WRITEI`: Escreve o valor inteiro na saída;
- `PUSHS "\n"`: Adiciona uma nova linha;
- `WRITES`: Escreve a nova linha na saída.

Vetor (Array): Se `NAME` for um vetor, o código gerado executa:

- Adiciona o delimitador inicial `"[ "` à saída com `PUSHS` e `WRITES`.
- Para cada índice do vetor, calcula o endereço do elemento com `PUSHGP`, `PUSHI`, e `PADD`;
- Lê o valor do elemento com `LOADN`;
- Escreve o valor com `WRITEI`;
- Adiciona espaços entre os elementos com `PUSHS` e `WRITES`;
- Finaliza com o delimitador `"]"` e uma nova linha.

Matriz (Matrix): Se `NAME` for uma matriz, o código gerado executa:

- Adiciona o delimitador inicial da linha `"[ "` com `PUSHS` e `WRITES`.
- Para cada coluna, calcula o endereço do elemento com `PUSHGP`, `PUSHI`, e `PADD`, considerando o índice da linha e o número de colunas;
- Lê o valor com `LOADN`;
- Escreve o valor com `WRITEI`;
- Adiciona espaços entre os valores com `PUSHS` e `WRITES`;
- Adiciona o delimitador final da linha `"]"` e uma nova linha.

5.9 Parser

```
parser = yacc.yacc()
parser.trackmap = dict()
parser.idLabel = 0
parser.memPointer = 0
```

Trata da inicialização das 3 componentes de controlo do projeto:

- **trackmap** - Dicionário que mantém informação sobre as variáveis
- **idLabel** - Contador de instruções de seleção e ciclos
- **memPointer** - Apontador para a próxima posição de memória livre

Capítulo 6

Codificação e Testes

6.1 Testes realizados e Resultados

Mostram-se a seguir alguns testes feitos (código introduzido) e os respectivos resultados obtidos:

6.1.1 Teste 1

Este teste declara e manipula variáveis atómicas do tipo inteiro, realiza operações aritméticas e lógicas, e usa instruções condicionais para imprimir valores com base em comparações.

```
var a, b, c;

a = 10;
b = 5;
c = a + b * 2;
print c;

if (a > b && c < 30) then {
    print a;
}

if (!(a == b)) then {
    print b;
}
```

De seguida, gerou-se o seguinte código **Assembly**:

```
PUSHI 0
START
PUSHI 15
STOREG 0
```

```
PUSHG 0
PUSHI 10
SUP
JZ l0
PUSHS "O número é maior que 10 \n"
WRITES
JUMP l0f
l0: NOP
PUSHS "O número não é maior que 10 \n"
WRITES
l0f: NOP
PUSHG 0
PUSHI 2
MOD
PUSHI 0
EQUAL
JZ l1
PUSHS "O número é par\n"
WRITES
JUMP l1f
l1: NOP
PUSHS "O número é ímpar\n"
WRITES
l1f: NOP
STOP
```

6.1.2 Teste 2

Este teste foca-se em atribuições algorítmicas, avaliando expressões aritméticas com diferentes operadores e precedência para atribuir valores a variáveis.

```
var x, y, z;

x = 5 + 3 * 2 - 1;
y = (10 / 2) % 3;
z = x + y;
print z;
```

De seguida, gerou-se o seguinte código **Assembly**:

```
PUSHI 0
PUSHI 0
PUSHI 0
START
PUSHI 5
PUSHI 3
PUSHI 2
PUSHI 1
```

```
SUB
MUL
ADD
STOREG 0
PUSHI 10
PUSHI 2
DIV
PUSHI 3
MOD
STOREG 1
PUSHG 0
PUSHG 1
ADD
STOREG 2
PUSHG 2
WRITEI
PUSHS "\n"
WRITES
STOP
```

6.1.3 Teste 3

Este teste demonstra a leitura de entrada do standard input e a escrita de saída para o standard output, interagindo com o utilizador para obter o seu nome e idade.

```
var x;
var y;
var result;

x = input;
y = input;

result = x + y;

print result;
```

De seguida, gerou-se o seguinte código **Assembly**:

```
PUSHI 0
PUSHI 0
PUSHI 0
START
READ
ATOI
STOREG 0
READ
ATOI
STOREG 1
```

```
PUSHG 0
PUSHG 1
ADD
STOREG 2
PUSHG 2
WRITEI
PUSHS "\n"
WRITES
STOP
```

6.1.4 Teste 4

Este teste exemplifica a estrutura de seleção if-then-else, executando diferentes blocos de código consoante uma condição seja verdadeira ou falsa, especificamente verificando se um número é maior que 10 e se é par ou ímpar.

```
var num;

num = 15;

if (num > 10) then {
    print "O número é maior que 10 \n";
} otherwise {
    print "O número não é maior que 10 \n";
}
if ((num % 2) == 0) then {
    print "O número é par\n";
} otherwise {
    print "O número é ímpar\n";
}
```

De seguida, gerou-se o seguinte código **Assembly**:

```
PUSHI 0
START
PUSHI 15
STOREG 0
PUSHG 0
PUSHI 10
SUP
JZ l0
PUSHS "O número é maior que 10 \n"
WRITES
JUMP l0f
l0: NOP
PUSHS "O número não é maior que 10 \n"
WRITES
l0f: NOP
```

```
PUSHG 0
PUSHI 2
MOD
PUSHI 0
EQUAL
JZ l1
PUSHS "O número é par\n"
WRITES
JUMP l1f
l1: NOP
PUSHS "O número é ímpar\n"
WRITES
l1f: NOP
STOP
```

6.1.5 Teste 5

Este teste cobre as estruturas de ciclo while-do e repeat-until, incluindo ciclos aninhados, demonstrando como controlar o fluxo de execução com base em condições e iterar através de um bloco de código múltiplas vezes.

```
var contador1, contador2;

contador1 = 0;
while (contador1 < 5) do {
    print "Iteração do ciclo while: ";
    print contador1;
    contador1 = contador1 + 1;

    contador2 = 0;
    repeat {
        print "  Ciclo repeat aninhado: ";
        print contador2;
        contador2 = contador2 + 1;
    } until (contador2 == 3);
}
```

De seguida, gerou-se o seguinte código **Assembly**:

```
PUSHI 0
PUSHI 0
START
PUSHI 0
STOREG 0
l1c: NOP
PUSHG 0
PUSHI 5
INF
```

```
JZ l1f
PUSHS "Iteração do ciclo while: \n"
WRITES
PUSHG 0
WRITEI
PUSHS "\n"
WRITES
PUSHG 0
PUSHI 1
ADD
STOREG 0
PUSHI 0
STOREG 1
l0c: NOP
PUSHS "  Ciclo repeat aninhado: \n"
WRITES
PUSHG 1
WRITEI
PUSHS "\n"
WRITES
PUSHG 1
PUSHI 1
ADD
STOREG 1
PUSHG 1
PUSHI 3
EQUAL
JZ l0c
JUMP l0f
l0f: NOP
JUMP l1c
l1f: NOP
STOP
```

6.1.6 Teste 6

Este teste demonstra a declaração e manipulação de um array unidimensional de inteiros, atribuindo valores aos seus elementos usando indexação direta e expressões, e iterando através do array para imprimir o seu conteúdo.

```
var x;
var arr[4];
var mat[2][2];

x = 1;

arr[0] = 1;
arr[1] = 222;
arr[2] = 3;
arr[3] = 1123;
```

```
mat[0][0] = 4;
mat[0][1] = 72;
mat[1][0] = 6;
mat[1][1] = 7;

print x;
print mat;
print arr;
```

De seguida, gerou-se o seguinte código **Assembly**:

```
PUSHI 0
PUSHN 4
PUSHN 4
START
PUSHI 1
STOREG 0
PUSHGP
PUSHI 1
PADD
PUSHI 0
PUSHI 1
STOREN
PUSHGP
PUSHI 1
PADD
PUSHI 1
PUSHI 222
STOREN
PUSHGP
PUSHI 1
PADD
PUSHI 2
PUSHI 3
STOREN
PUSHGP
PUSHI 1
PADD
PUSHI 3
PUSHI 1123
STOREN
PUSHGP
PUSHI 5
PADD
PUSHI 0
PUSHI 2
MUL
PUSHI 0
ADD
PUSHI 4
STOREN
PUSHGP
```

```
PUSHI 5
PADD
PUSHI 0
PUSHI 2
MUL
PUSHI 1
ADD
PUSHI 72
STOREN
PUSHGP
PUSHI 5
PADD
PUSHI 1
PUSHI 2
MUL
PUSHI 0
ADD
PUSHI 6
STOREN
PUSHGP
PUSHI 5
PADD
PUSHI 1
PUSHI 2
MUL
PUSHI 1
ADD
PUSHI 7
STOREN
PUSHG 0
WRITEI
PUSHS "\n"
WRITES
PUSHS "[ "
WRITES
PUSHGP
PUSHI 5
PADD
PUSHI 0
LOADN
WRITEI
PUSHS " "
WRITES
PUSHGP
PUSHI 5
PADD
PUSHI 1
LOADN
WRITEI
PUSHS " "
WRITES
PUSHS "]\n"
WRITES
PUSHS "[ "
```

```
WRITES
PUSHGP
PUSHI 5
PADD
PUSHI 2
LOADN
WRITEI
PUSHS " "
WRITES
PUSHGP
PUSHI 5
PADD
PUSHI 3
LOADN
WRITEI
PUSHS " "
WRITES
PUSHS "]\n"
WRITES
PUSHS "[ "
WRITES
PUSHGP
PUSHI 1
PADD
PUSHI 0
LOADN
WRITEI
PUSHS " "
WRITES
PUSHGP
PUSHI 1
PADD
PUSHI 1
LOADN
WRITEI
PUSHS " "
WRITES
PUSHGP
PUSHI 1
PADD
PUSHI 2
LOADN
WRITEI
PUSHS " "
WRITES
PUSHGP
PUSHI 1
PADD
PUSHI 3
LOADN
WRITEI
PUSHS " "
WRITES
PUSHS "]"
```

```
WRITES
PUSHS "\n"
WRITES
STOP
```

6.1.7 Teste Sequência de Fibonacci

```
var n, a, b, temp;
print "Digite um numero para calcular Fibonacci: \n";
n = input;
a = 0;
b = 1;

print "Sequencia de Fibonacci:\n";

if (n == 0) then {
    print a;
} otherwise {
    if (n == 1) then {
        print a;
        print b;
    } otherwise {
        print a;
        print b;
        while (n > 1) do {
            temp = a + b;
            a = b;
            b = temp;
            print temp;
            n = n - 1;
        }
    }
}
```

De seguida, gerou-se o seguinte código **Assembly**:

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
PUSHS "Digite um numero para calcular Fibonacci: \n"
WRITES
READ
ATOI
STOREG 0
PUSHI 0
STOREG 1
PUSHI 1
```

```
STOREG 2
PUSHS "Sequencia de Fibonacci:\n"
WRITES
PUSHG 0
PUSHI 0
EQUAL
JZ l2
PUSHG 1
WRITEI
PUSHS "\n"
WRITES
JUMP l2f
l2: NOP
PUSHG 0
PUSHI 1
EQUAL
JZ l1
PUSHG 1
WRITEI
PUSHS "\n"
WRITES
PUSHG 2
WRITEI
PUSHS "\n"
WRITES
JUMP l1f
l1: NOP
PUSHG 1
WRITEI
PUSHS "\n"
WRITES
PUSHG 2
WRITEI
PUSHS "\n"
WRITES
l0c: NOP
PUSHG 0
PUSHI 1
SUP
JZ l0f
PUSHG 1
PUSHG 2
ADD
STOREG 3
PUSHG 2
STOREG 1
PUSHG 3
STOREG 2
PUSHG 3
WRITEI
PUSHS "\n"
WRITES
PUSHG 0
PUSHI 1
```

```
SUB
STOREG 0
JUMP l0c
l0f: NOP
l1f: NOP
l2f: NOP
STOP
```

Capítulo 7

Conclusão

Este relatório descreve o processo de implementação de um compilador, com o propósito de processar uma linguagem fictícia, inventada por nós, alunos.
Este processo passou por todas as fases de desenvolvimento propostas na UC de Processamento de Linguagens e Compiladores, pelo que se considera que, de forma geral, corresponde aos objetivos previamente definidos.

O projeto foi um bom desafio, além de ser uma boa forma de aplicarmos os conhecimentos adquiridos ao longo do semestre, também nos levou a perceber que, no que toca a compiladores, é necessário uma boa definição das regras a seguir.
Apesar de parecer um conceito relativamente simples, provou ser bastante desafiante implementar os pontos pedidos, pois por vezes encontramos inconsistências que nos forçaram a dar um passo atrás e repensar a abordagem ao problema.

A nosso ver, os pontos requeridos foram cumpridos, até, em parte, conseguimos superar os mesmos, por exemplo, o nosso projeto permite a escrita de **Strings** no *Standard Output*.
Porém, existe algum espaço para trabalho futuro, por exemplo, com algumas possíveis adições que fariam a nossa linguagem de programação parecer mais real e abrir mais possibilidades.
Mais concretamente, poderíamos ter implementado mais formas de controlo de erros, de forma a um utilizador ter mais informação proveniente do nosso código e não do próprio *Python*. Poderia também incluir definição e invocação de funções sem parâmetros, funções com nomes precisos de forma a facilitar a utilização.

Apêndice A

Código Lex

```
import ply.lex as lex

#### Definição de Palavras Reservadas ####

reserved = {
    'print': 'PRINT',
    'input': 'INPUT',
    'while': 'WHILE',
    'do': 'DO',
    'repeat': 'REPEAT',
    'until': 'UNTIL',
    'if': 'IF',
    'then': 'THEN',
    'otherwise': 'OTHERWISE',
    'eq': 'EQ',
    'neq': 'NEQ',
    'moreeq': 'MOREEQ',
    'lesseq': 'LESSEQ',
    'and': 'AND',
    'or': 'OR',
}

#### Definição de Tokens ####

tokens = [
    'NAME',
    'VAR',
    'NUM',
    'STRING',
] + list(reserved.values())

#### Definição de Literais ####

literals = [';',':',',','=','{','}','|','"','','[',']','(',')',';','+','-','*','/','%
    ','>','<','!']


#### Regras de Expressões Regulares ####
```

```
t_EQ = r'\=\='
t_NEQ = r'\!\='
t_MOREEQ = r'\>\='
t_LESSEQ  = r'\<\='
t_AND = r'\&\&'
t_OR = r'\|\|'

#### Regras de Tokens ####

def t_VAR(t):
    r'var'
    return t

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

def t_NAME(t):
    r'[a-zA-ZÀ-ÿ_][a-zA-ZÀ-ÿ0-9_]*'
    t.type = reserved.get(t.value, 'NAME')
    return t

def t_STRING(t):
    r'"([^"\\]|\\.)*"'
    t.value = t.value.strip('"')
    return t

#### Gestão de Linhas Novas ####

def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

#### Gestão de Erros ####

def t_error(t):
    print(f"Illegal character '{t.value[0]}' at line {t.lineno}")
    t.lexer.skip(1)

#### Ignorar Espaços em Branco ####

t_ignore  = ' \r\t'

def t_COMMENT(t):
    r'\#.*'
    pass

#### Inicialização do Lexer ####

lexer = lex.lex()
```

Apêndice B

Código Yacc

```
from LexTP import tokens
import ply.yacc as yacc
import sys
import os

#### Estrutura Programa ####

def p_Program(p):
    "Program : Header Code"
    p[0] = p[1] + "START\n" + p[2] + "STOP\n"


#### Estrutura do Header ####

def p_WOHeader(p):
    "Program : Code"
    p[0] = "START\n" + p[1] + "STOP\n"

def p_MultHeader(p):
    "Header : Header Decl"
    p[0] = p[1] + p[2]

def p_SingleHeader(p):
    "Header : Decl"
    p[0] = p[1]


#### Declaration [Int - Array - Matrix] ####

def p_IntDecl(p):
    "Decl : VAR NameList ';'"
    for name in p[2]:
        if name not in p.parser.trackmap:
            p.parser.trackmap.update({name: p.parser.memPointer})
            p[0] = (p[0] or "") + "PUSHI 0\n"
            p.parser.memPointer += 1
        else:
            raise Exception(f"Variable {name} already declared.")
```

```
def p_NameList(p):
    """NameList : NAME
                | NameList ',' NAME"""
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[3]]

def p_ArrayDecl(p):
    "Decl : VAR NAME '[' NUM ']' ';'"
    if p[2] not in p.parser.trackmap:
        p.parser.trackmap.update({p[2]: (p.parser.memPointer, int(p[4]))})
        memSpace = int(p[4])
        p[0] = f"PUSHN {memSpace}\n"
        p.parser.memPointer += int(p[4])
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[2]} is already declared.")

def p_MatrixDecl(p):
    "Decl : VAR NAME '[' NUM ']' '[' NUM ']' ';'"
    if p[2] not in p.parser.trackmap:
        p.parser.trackmap.update({p[2]: (p.parser.memPointer, int(p[4]), int(p[7]))})
        memSpace = int(p[4]) * int(p[7])
        p[0] = f"PUSHN {memSpace}\n"
        p.parser.memPointer += memSpace
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[2]} is already declared.")


#### Code Structure ####

def p_MultCode(p):
    "Code : Code Codes"
    p[0] = p[1] + p[2]

def p_SingleCode(p):
    "Code : Codes"
    p[0] = p[1]

def p_Codes(p):
    """Codes : Conditions
             | WhileDo
             | RepeatUntil
             | Assign
             | Print
    """
    p[0] = p[1]


#### Cycles and Conditions ####

def p_CondIfThen(p):
    "Conditions : IF '(' Condition ')' THEN '{' Code '}'"
```

```
    p[0] = p[3] + f"JZ l{p.parser.idLabel}\n" + p[7] + f"l{p.parser.idLabel}:
    p.parser.idLabel += 1

def p_CondIfThenOtherwise(p):
    "Conditions : IF '(' Condition ')' THEN '{' Code '}' OTHERWISE '{' Code '}'"
    p[0] = p[3] + f"JZ l{p.parser.idLabel}\n" + p[7] + f"JUMP l{p.parser.idLabel}f\nl
       {p.parser.idLabel}: NOP\n" + p[11] + f"l{p.parser.idLabel}f: NOP\n"
    p.parser.idLabel += 1

def p_WhileDo(p):
    "WhileDo : WHILE '(' Condition ')' DO '{' Code '}'"
    p[0] = f"l{p.parser.idLabel}c: NOP\n" + p[3] + f"JZ l{p.parser.idLabel}f\n" + p
       [7] + f"JUMP l{p.parser.idLabel}c\nl{p.parser.idLabel}f: NOP\n"
    p.parser.idLabel += 1

def p_RepeatUntil(p):
    "RepeatUntil : REPEAT '{' Code '}' UNTIL '(' Condition ')' ';' "
    p[0] = (
        f"l{p.parser.idLabel}c: NOP\n" + p[3]+ p[7]+ f"JZ l{p.parser.idLabel}c\n"+ f"
           JUMP l{p.parser.idLabel}f\n"+ f"l{p.parser.idLabel}f: NOP\n"
    )
    p.parser.idLabel += 1


#### Assigning ####

def p_ExpressionAssign(p):
    "Assign : NAME '=' Expr ';'"  #exemplo : b = 2
    if p[1] in p.parser.trackmap:
        var = p.parser.trackmap.get(p[1])
        if type(var) == int:
            p[0] = p[3] + f"STOREG {var}\n" # Armazena o resultado da expressão na
               variável.
        else:
            raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not an integer.")  #
               Verifica se a variável é um inteiro.
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")

def p_ArrayAssign(p):
     "Assign : NAME '[' Expr ']' '=' Expr ';'"   # Array tem de permitir Expr para
        ver índices (letras) nos ciclos
     if p[1] in p.parser.trackmap:
          varInfo = p.parser.trackmap.get(p[1])
          if len(varInfo) == 2:
               p[0] = f'PUSHGP\nPUSHI {varInfo[0]}\nPADD\n' + p[3] + p[6] + 'STOREN\
                  n'  # Permite índices expressivos nos ciclos.
          else:
               raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not an array.")
     else:
          raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")

def p_MatrixAssign(p): # Nome[Expr][Expr] = Expr
     "Assign : NAME '[' Expr ']' '[' Expr ']' '=' Expr ';'"
```

```
     if p[1] in p.parser.trackmap:
            varInfo = p.parser.trackmap.get(p[1])
            if len(varInfo) == 3:
                  p[0] = f'PUSHGP\nPUSHI {varInfo[0]}\nPADD\n{p[3]}PUSHI {varInfo
                      [2]}\nMUL\n{p[6]}ADD\n{p[9]}STOREN\n'
            else:
                  raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not a matrix.")
     else:
          raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")


#### Expressions ####

def p_Expr_condition(p):
    'Expr : Condition'
    p[0] = p[1]

def p_Expr_Variable(p):
    'Expr : Variable'
    p[0] = p[1]

def p_expression_number(p):
    "Expr : NUM"
    p[0] = f'PUSHI {p[1]}\n'

def p_Expr_OP(p):
    """Expr : Expr "+" Expr
            | Expr "-" Expr
            | Expr "*" Expr
            | Expr "/" Expr
            | Expr "%" Expr"""

    if (p[2] == '+'):
        p[0] = p[1] + p[3] + "ADD \n"
    elif (p[2] == '-'):
        p[0] = p[1] + p[3] + "SUB \n"
    elif (p[2] == '*'):
        p[0] = p[1] + p[3] + "MUL \n"
    elif (p[2] == '/'):
        p[0] = p[1] + p[3] + "DIV \n"
    elif (p[2] == '%'):
        p[0] = p[1] + p[3] + "MOD \n"


def p_condLog(p):
    """Condition : Expr EQ Expr
                 | Expr NEQ Expr
                 | '!' Expr
                 | Expr '>' Expr
                 | Expr MOREEQ Expr
                 | Expr '<' Expr
                 | Expr LESSEQ Expr
                 | Expr AND Expr
                 | Expr OR Expr"""
```

```
    if (p[2] == "=="):
        p[0] = p[1] + p[3] + "EQUAL \n"
    elif (p[2] == "!="):
          p[0] = p[1] + p[3] + "EQUAL\nNOT \n"
    elif (p[1] == '!'):
        p[0] = p[2] + "NOT \n"
    elif (p[2] == '>'):
        p[0] = p[1] + p[3] + "SUP \n"
    elif (p[2] == ">="):
        p[0] = p[1] + p[3] + "SUPEQ \n"
    elif (p[2] == '<'):
        p[0] = p[1] + p[3] + "INF \n"
    elif (p[2] == "<="):
        p[0] = p[1] + p[3] + "INFEQ \n"
    elif (p[2] == "&&"):
        p[0] = p[1] + p[3] + 'ADD\nPUSHI 2\nEQUAL\n'
    elif (p[2] == "||"):
        p[0] = p[1] + p[3] + "ADD\nPUSHI 1\nSUPEQ\n"
    elif (p[1] == "Var"):
        p[0] = p[1]

def p_Expr_base(p):
    "Expr : '(' Expr ')'"
    p[0] = p[2]

def p_condition_base(p):
    "Condition : '(' Condition ')'"
    p[0] = p[2]


#### Acessing Vars (Num - Array - Matrix ) ####

def p_VarNum(p):
    "Variable : NAME"
    if p[1] in p.parser.trackmap:
        var = p.parser.trackmap.get(p[1])
        if type(var) == int:
            p[0] = f"PUSHG {var}\n"
        else:
            raise TypeError(f"Line {p.lineno(1)}, {p[1]} is not an integer.")
    else:
        raise Exception(f"Line {p.lineno(1)}, {p[1]} is not declared.")

def p_VarArray(p):
    "Variable : NAME '[' Expr ']'"
    if p[1] in p.parser.trackmap:
        varInfo = p.parser.trackmap.get(p[1])
        if len(varInfo) == 2:
            p[0] = f'PUSHGP\nPUSHI {varInfo[0]}\nPADD\n' + p[3] + 'LOADN\n'
        else:
            raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not an array.")
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")
```

```
def p_VarMatrix(p):
    "Variable : NAME '[' Expr ']' '[' Expr ']'"
    if p[1] in p.parser.trackmap:
        varInfo = p.parser.trackmap.get(p[1])
        if len(varInfo) == 3:
            p[0] = f'PUSHGP\nPUSHI {varInfo[0]}\nPADD\n' + p[3] + f'PUSHI {varInfo
                [2]}\nMUL\n' + p[6] + 'ADD\n' + 'LOADN\n'
        else:
            raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not a matrix.")
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")


#### Error ####

def p_error(p):
    print(f"Syntax error: token {p.value} on line {p.lineno}.")


#### Input (Assignments) ####

def p_Input_Array(p):
    "Assign : NAME '[' Expr ']' '=' INPUT ';'"
    if p[1] in p.parser.trackmap:
        var = p.parser.trackmap.get(p[1])
        if len(var) == 2:
            p[0] = f'PUSHGP\nPUSHI {var[0]}\nPADD\n' + p[3] + f'READ\nATOI\nSTOREN\n'
        else:
            raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not an array.")
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")

def p_Input_Matrix(p):
    "Assign : NAME '[' Expr ']' '[' Expr ']' '=' INPUT ';'"
    if p[1] in p.parser.trackmap:
        var = p.parser.trackmap.get(p[1])
        if len(var) == 3:
            p[0] = f'PUSHGP\nPUSHI {var[0]}\nPADD\n{p[3]}PUSHI {var[2]}\nMUL\n{p[6]}
                ADD\nREAD\nATOI\nSTOREN\n'
        else:
            raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not a matrix.")
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")

def p_Input_Var(p):
    "Assign : NAME '=' INPUT ';'"
    if p[1] in p.parser.trackmap:
        var = p.parser.trackmap.get(p[1])
        if type(var) == int:
            p[0] = f'READ\nATOI\nSTOREG {var}\n'
        else:
            raise TypeError(f"Line{p.lineno(2)}, {p[1]} is not an integer.")
    else:
```

```
        raise Exception(f"Line{p.lineno(2)}, {p[1]} is not declared.")


#### Print ####

def p_Print(p):
    "Print : PRINT NAME ';'"
    if p[2] in p.parser.trackmap:
        var = p.parser.trackmap.get(p[2])

        if type(var) == tuple:
            if len(var) == 2:
                array = f'PUSHS "[ "\nWRITES\n'
                for i in range(var[1]):
                    array += f'PUSHGP\nPUSHI {var[0]}\nPADD\nPUSHI {i}\nLOADN\nWRITEI
                        \nPUSHS " "\nWRITES\n'
                array += f'PUSHS "]"\nWRITES\n'
                p[0] = array + 'PUSHS "\\n"\nWRITES\n'

            elif len(var) == 3:
                matrix = ""
                for i in range(var[1]):
                    matrix += f'PUSHS "[ "\nWRITES\n'
                    for j in range(var[2]):
                        matrix += f'PUSHGP\nPUSHI {var[0]}\nPADD\nPUSHI {var[2] * i +
                            j}\nLOADN\nWRITEI\nPUSHS " "\nWRITES\n'
                    matrix += 'PUSHS "]\\n"\nWRITES\n'
                p[0] = matrix
        else:
            p[0] = f'PUSHG {var}\nWRITEI\nPUSHS "\\n"\nWRITES\n'
    else:
        raise Exception(f"Line{p.lineno(2)}, {p[2]} is not declared.")

def p_PrintString(p):
    "Print : PRINT STRING ';'"
    p[0] = f'PUSHS "{p[2]}"\nWRITES\n'


#### Parser Initializer ####

parser = yacc.yacc()
parser.trackmap = dict()
parser.idLabel = 0
parser.memPointer = 0


#### Main ####

if len(sys.argv) > 1:
    input_file = sys.argv[1]
    if not os.path.exists("Outputs"):
        os.makedirs("Outputs")
    output_file = os.path.join("Outputs", "Assembly_" + os.path.basename(input_file))
```

```
    with open(input_file, 'r') as file:
        assembly = parser.parse(file.read())
        if assembly:
            with open(output_file, 'w') as output:
                output.write(assembly)
                print(f"{input_file} compiled successfully!\nCheck the output in {
                    output_file}.")
        else:
            print("Empty!")
else:
    # Interactive mode
    line = input(">")
    while line != "\n":
        print(parser.parse(line))
        line = input(">")
```