Processamento de Linguagens (3º ano de MIEI)

Trabalho Prático 2

Relatório de Desenvolvimento

Sofia Santos
(a89615)

Carolina Vila Chã
(a89495)

1 de julho de 2021

Resumo

Este relatório aborda o desenvolvimento de um compilador usando módulos de gramáticas tradutoras do Python, no contexto do 2º trabalho prático da UC Processamento de Linguagens.

Conteúdo

Capítulo 1

Introdução

Tal como nós, as nossas máquinas possuem a capacidade de realizar a análise sintática de texto, decompondo um conjunto de dados de entrada em unidades estruturais, a partir de uma gramática tradutora. Este método, conhecido por *parsing*, é precedido de uma análise léxica, na qual uma sequência de caracteres é convertida numa sequência de *tokens*.

Neste trabalho iremos, com o auxílio da linguagem de programação Python e do módulo `ply`, desenvolver um programa capaz de fazer a análise léxica e sintática de uma linguagem imperativa desenvolvida por nós. Mais especificamente, o nosso programa irá converter código desenvolvido na nossa linguagem para código máquina que será depois corrido numa máquina de stack virtual.

Foram-nos dadas duas opções de funcionalidades opcionais que a nossa linguagem desenvolvida deve suportar. Escolhemos a opção relativa a variáveis do tipo *array*.

Capítulo 2

Enunciado

Transcrição do enunciado do projeto, fornecido pelos docentes da cadeira:

Pretende-se que comece por definir uma linguagem de programação imperativa simples, a seu gosto. Apenas deve ter em consideração que essa linguagem terá de permitir:

- declarar variáveis atómicas do tipo inteiro, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas.
- efetuar instruções algorítmicas básicas como a atribuição do valor de expressões numéricas a variáveis.
- ler do standard input e escrever no standard output.
- efetuar instruções condicionais para o controlo do fluxo de execução.
- declarar variáveis atómicas do tipo inteiro, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas.

Adicionalmente deve ainda suportar, à sua escolha, uma das duas funcionalidades seguintes:

- declarar e manusear variáveis estruturadas do tipo array (a 1 ou 2 dimensões) de inteiros, em relação aos quais é apenas permitida a operação de indexação (índice inteiro).
- definir e invocar subprogramas sem parâmetros mas que possam retornar um resultado do tipo inteiro.

Como é da praxe neste tipo de linguagens, as variáveis deverão ser declaradas no início do programa e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explicitado, o valor da variável após a declaração é 0 (zero). Desenvolva, então, um compilador para essa linguagem com base na GIC criada acima e com recurso aos módulos Yacc/ Lex do PLY/Python. O compilador deve gerar pseudo-código, Assembly da Máquina Virtual VM.
Muito Importante:
Para a entrega do TP deve preparar um conjunto de testes (programas-fonte escritos na sua linguagem) e mostrar o código Assembly gerado bem como o programa a correr na máquina virtual VM. Esse conjunto terá de conter, no mínimo, os 4 primeiros exemplos abaixo e um dos 2 últimos conforme a sua escolha acima:

- ler 4 números e dizer se podem ser os lados de um quadrado.
- ler um inteiro N, depois ler N números e escrever o menor deles.

- ler N (constante do programa) números e calcular e imprimir o seu produtório.
- contar e imprimir os números ímpares de uma sequência de números naturais.
- ler e armazenar N números num array; imprimir os valores por ordem inversa.
- invocar e usar num programa ser uma função 'potencia()', que começa por ler do input a base B e o expoente E e retorna B^E.

Capítulo 3

Linguagem Desenvolvida

A linguagem que desenvolvemos para este trabalho prático, à qual demos o nome de Linguagem Genérica Baseada nas Típicas Imperativas, ou LGBTI, para poder identificar facilmente os seus ficheiros, possui uma sintaxe muito semelhante à da linguagem de programação C, com algumas simplificações.

3.1 Estrutura dos programas

Um programa desenvolvido com a nossa linguagem de programação deve seguir uma estrutura própria. A primeira parte do programa destina-se a declarações de variáveis, enquanto que a segunda parte contém os restantes comandos implementados na linguagem. Não é permitido declarar novas variáveis depois de executar outros comandos, nem é possível redeclarar variáveis ou usar variáveis que não tenham sido declaradas. Qualquer comando deve terminar com ';'.

3.2 Declarações

A declaração de variáveis é feita de forma muito semelhante à usada em outras linguagens imperativas. Primeiro especificamos o tipo de dados que queremos que a variável armazene, depois o nome da variável em si, e depois o seu valor. Uma variável pode ser do tipo inteiro (*int*), vírgula flutuante (*float*), ou *string*. Podemos ainda declarar mais do que uma variável na mesma linha, assim como declarar um array de valores inteiros, com 1 ou 2 dimensões.

Alguns exemplos de declarações:

```
int x = 3;
float y = 6.2;
string z = "Hello world!";
int a, b = -4, c;
int d[3] = [1,2,3];
int e[5][4];
```

Se uma variável do tipo inteiro ou vírgula flutuante for declarada sem um valor inicial, esta assume um valor nulo, ou seja, 0. O mesmo se aplica a *arrays*.

3.3 Input/Output

Para obter *input* do utilizador é possível usar a função `input(t)`, que recebe como argumento o texto a imprimir antes de obter o input, e devolve uma *string* com o valor introduzido.

No sentido inverso, a função `print(x)` recebe o valor a mostrar, que pode ser de qualquer tipo, e imprime-o no terminal do utilizador.

Existe ainda a função `println(x)` que faz o mesmo que a função `print()`, mas acrescenta um caracter de mudança de linha ('\n') no final do output.

Exemplo de utilização destes comandos:

```
string nome = input("Introduza o seu nome: ");
print("O seu nome é ");
println(nome);
```

Output respetivo no terminal:

```
Introduza o seu nome: Marcelo Rebelo de Sousa
O seu nome é Marcelo Rebelo de Sousa
```

3.4 Operações aritméticas, relacionais, lógicas e de conversão

Tal como seria de esperar, é possível realizar as habituais operações de adição, subtração, multiplicação e divisão, tanto com valores inteiros como com valores de vírgula flutuante. É ainda possível realizar operações de módulo, mas apenas com valores inteiros. A ordem das operações e dos parênteses é respeitada, como seria de esperar.

Depois de executar o comando `int x = 2 + 3 * 4;`, a variável `x` terá o valor `14`, enquanto que o comando `float x = 10.0 / (2.0 + 6.0) - 8.0` fará com que `x` contenha o valor `-6.75`.

Para simplificar algumas operações, é possível ainda fazer somas e subtrações do tipo `x++` e `x--`, tal como em várias outras linguagens de programação.

Se um dos operandos for um valor em vírgula flutuante e o outro for inteiro, será feita uma conversão implícita do operando inteiro para vírgula flutuante.

Podemos ainda realizar operações lógicas, tais como `a > b`, `a <= b` ou `a == b`, cujo resultado é sempre um valor inteiro. Estas operações lógicas podem ser encadeadas com os operadores `&&/and`, `||/or` e `!/not`.

As funções `int(x)` e `float(x)` permitem-nos fazer a conversão explícita de uma variável para o tipo inteiro ou vírgula flutuante, respetivamente. Se o argumento for uma *string*, é feita a extração de um valor da mesma. Estas funções, encadeadas com a função `input()`, dão-nos a capacidade de obter *input* do utilizador em formato numérico (ex: `int idade = int(input("Introduza a sua idade: "));`).

3.5 Instruções condicionais

A sintaxe das instruções condicionais é idêntica à usada pela maioria das linguagens de programação. Primeiro temos a *keyword* "if", depois uma condição e por fim, entre chavetas, um conjunto de instruções a realizar caso a condição seja verdadeira. Aqui, a condição pode estar entre parênteses ou não, ao contrário do que acontece em linguagens como C. É possível ter ainda um conjunto de instruções a realizar caso a condição não seja verdadeira, delimitadas pelo operador "else". Quem tiver a ambição para tal pode ainda encadear estas instruções condicionais, desta forma: `if (...) { ... } else if (...) { ... } else if (...) {...} ...`

Apresentamos a seguir um exemplo mais concreto do uso destes operadores:

```
int idade;
println("Descubra se pode conduzir!");
idade = int(input("Introduza a sua idade: "));
if (idade < 18) {
    println("Não pode conduzir.");
}
else if idade < 75 {
    println("Pode conduzir, desde que tenha carta de condução.");
}
else {
    println("Ainda pode conduzir, mas deve ter bastante cuidado se o fizer.");
}
```

3.6 Instruções cíclicas

A nossa linguagem possui dois tipos de instruções cíclicas. O primeiro, identificado pela *keyword* "while", executa um conjunto de instruções enquanto que a condição fornecida seja verdadeira. O segundo, identificado pela *keyword* "for", também executa instruções enquanto uma dada condição seja verdadeira, a diferença é que este tipo de ciclo tem associado a ele uma operação que é realizada a cada iteração. Geralmente, esta operação é usada para incrementar ou decrementar uma variável associada ao ciclo, mas tal não é obrigatório. Podemos ver assim que um ciclo "for"não passa de uma versão mais específica de um ciclo "while". Segundo o número do nosso grupo, poderíamos ter implementado apenas o ciclo `for`, mas como a sua implementação é muito parecida implementámos também o ciclo `while`.
Exemplo de um ciclo "for":

```
int r = 1;
int i;
for(i = 1 ; i < 5; i++) {
    r = r * i;
}
```

Capítulo 4

Regras de Tradução

De modo a organizar melhor as nossas estruturas e obter um resultado final mais "estruturado", o Lexer e o Parser do nosso programa estão definidos em classes. Esta implementação não é muito diferente da que foi usada nas aulas de Processamento de Linguagens, apenas nos permite ter um certo "encapsulamento"e organização, algo vital para um projeto de maior dimensão.

4.1 Lexer

É possível consultar o código para o Lexer na sua totalidade no Anexo C.

Para além da definição dos obrigatórios *tokens*, e da definição dos *literals*, foram também definidas palavras reservadas (*reserved*). As palavras reservadas permitem evitar conflitos na interpretação do texto. Quando o Lexer lê um conjunto de caracteres procura por esse conjunto exato num dicionário de palavras reservadas. Se encontrar uma correspondência, atribui ao *token* lido o tipo correspondente à chave encontrada. Caso contrário, o *token* irá corresponder a uma variável. Sem esta implementação, se tivéssemos uma variável chamada "interesse", por exemplo, o Lexer iria interpretar as três primeiras letras como a *keyword* para definir um valor inteiro, o que não é o comportamento pretendido neste caso.

Ainda na regra das variáveis (`t_ID`), temos um mecanisco que deteta se a variável introduzida existe no programa ou não. Isto permite-nos saber qual o tipo da variável antes de entrar no Parser, o que reduz bastante o trabalho do mesmo, visto que não temos de nos preocupar em realizar controlo de tipos, isto é feito por nós automaticamente. Para obter esta informação, o Lexer e o Parser partilham um dicionário (`self.fp`) que contém todas as variáveis em uso pelo programa e o seu tipo.

```
def t_ID(self, t):
    r"[a-zA-Z_][a-zA-Z0-9_\']*"
    t.type = Lexer.reserved.get(t.value,"ID")
    v = self.fp.get(t.value, None)
    if v is not None:
        vartype = v[1]
        if vartype == INT: t.type = "VAR"
        elif vartype == FLOAT: t.type = "VARF"
        elif vartype == STRING: t.type = "VARS"
        elif vartype == ARRAY: t.type = "VARA"
    return t
```

De resto, a única que vale a pena realçar é que, para além do que foi pedido no enunciado do trabalho prático, implementámos operações com números em vírgula flutuante. Para tal, criámos uma regra no Lexer que nos permite ler estes valores, em formato decimal ou em notação científica.

4.2 Parser

É possível consultar o código para o Parser na sua totalidade no Anexo D.

4.2.1 Estrutura Básica

Tal como referimos no capítulo anterior, a nossa linguagem deve conter as declarações de variáveis antes do resto das instruções. Como tal, as regras base do nosso programa são as seguintes:

```
def p_Program(self, p):
    "Program : Attribs Commands"
    p[0] = p[1] + "start\n" + p[2] + "stop\n"

def p_Program_noAttribs(self, p):
    "Program : Commands"
    p[0] = "start\n" + p[1] + "stop\n"

def p_Program_noCommands(self, p):
    "Program : Attribs"
    print("Error: no instructions found.")
    raise SyntaxError

def p_Program_error(self, p):
    "Program : error"
    print("Aborting...")
    exit(1)
```

A quarta regra funciona como um "catch-all"para qualquer erro que possa ocorrer durante a compilação do programa, abortando o processo em vez de tentar continuar. Temos algumas mensagens de erros mais específicas noutras partes do compilador, mas esta é a que se encarrega de interromper a execução.

O valor de `p[0]` na primeira regra irá corresponder ao programa final compilado, pronto a ser lido pela VM. Iremos explicar o que fazemos com este valor mais à frente.

Declaração de Variáveis

Como foi possível ver acima, o conjunto das declarações do programa corresponde ao não-terminal "Attribs", que pode ser decomposto em atribuições individuais.

```
def p_Attribs(self, p):
    "Attribs : Attribs Attrib"
    p[0] = p[1] + p[2]
```

```
def p_Attribs_single(self, p):
    "Attribs : Attrib"
    p[0] = p[1]
```

Cada atribuição contém o identificador do tipo da variável, o seu nome e opcionalmente um valor inicial.
Para não tornar o relatório demasiado repetitivo, apenas iremos mostrar como funciona a atribuição para variáveis inteiras e arrays unidimensionais, sendo que é possível consultar as outras no Anexo D e todas funcionam de forma semelhante.

```
def p_Attrib(self, p):
    "Attrib : INTKW AttribsInt ';'"
    p[0] = p[2]

def p_AttribsInt(self, p):
    "AttribsInt : AttribsInt ',' AttribInt"
    p[0] = p[1] + p[3]

def p_AttribsInt_single(self, p):
    "AttribsInt : AttribInt"
    p[0] = p[1]

def p_AttribInt(self, p):
    """AttribInt : ID '=' Expression
                 | ID"""
    i = self.stack_size
    if p[1] not in self.fp:
        self.fp[p[1]] = (i,INT)
        self.stack_size += 1
    else:
        print(f"Error in line {p.lineno(1)}: variable {p[1]} was previously declared.")
        raise SyntaxError
    p[0] = p[3] if len(p) > 2 else 'pushi 0\n'
```

Vemos assim como é que o Parser lida com várias declarações na mesma linha, por exemplo, ou com declarações sem valor inicial dado.
Tal como referimos na secção do Lexer, as duas classes partilham um dicionário que contém todas as variáveis do programa a compilar, os seus tipos e a sua posição na stack da VM. Para além disso, a variável `self.stack_size` armazena o tamanho atual deste dicionário (é necessário pois um array apenas ocupa uma entrada do dicionário, mas a estrutura em si ocupa n lugares na stack, logo não podemos simplesmente usar o tamanho do dicionário como indicador do tamanho da stack).
O não-terminal "Expression"diz respeito a um valor inteiro, mas iremos abordá-lo em detalhe mais à frente.

```
def p_Attrib_Array(self, p):
    "Attrib : INTKW ID '[' INT ']' ';'"
```

```
    if p[2] not in self.fp:
        self.fp[p[2]] = (self.stack_size,ARRAY,p[4])
        self.stack_size += p[4]
    else:
        print(f"Error in line {p.lineno(2)}: variable {p[2]} was previously declared.")
        raise SyntaxError
    p[0] = f"pushn {p[4]}\n"

def p_Array(self, p):
    "Array : '[' Elems ']' "
    p[0] = p[2]

def p_Elems(self, p):
    "Elems : Elems ',' INT"
    p[0] = p[1]
    p[0].append(p[3])

def p_Elems_single(self, p):
    "Elems : INT"
    p[0] = [p[1]]

def p_Attrib_Array_elems(self, p):
    "Attrib : INTKW ID '[' INT ']' '=' Array ';'"
    "Attrib : INTKW ID '[' Empty ']' '=' Array ';'"
    if not p[4]: p[4] = len(p[7])
    if p[2] not in self.fp:
        self.fp[p[2]] = (self.stack_size,ARRAY,p[4])
        self.stack_size += p[4]
    else:
        print(f"Error in line {p.lineno(2)}: variable {p[2]} was previously declared.")
        raise SyntaxError

    if len(p[7]) != p[4]:
        print(f"Error in line {p.lineno(2)}: number of array elements different from the
        ↪  defined value.")
        raise SyntaxError
    p[0] = ""
    for elem in p[7]:
        p[0] += f"pushi {elem}\n"
```

Declarar um array vazio é simples, apenas nos devemos preocupar em armazenar o tamanho do mesmo no nosso dicionário de variáveis e devolver a instrução para a VM com o valor correto.

Por outro lado, para definir um array com valores iniciais, temos primeiro de os juntar todos numa lista e depois percorrer a mesma, criando instruções para a VM por cada valor na lista. Podemos também verificar se a quantidade de valores dados corresponde ao valor dado na declaração, se for especificado um valor, e se diferirem levantar um erro de sintaxe.

A regra "Empty", definida da seguinte forma:

```
def p_Empty(self, p):
    "Empty : "
    pass
```

é extremamente útil para podermos ter valores opcionais nas nossas regras sem precisarmos de definir duas funções diferentes, pois assim garantimos que os outros símbolos mantêm a sua posição.

Operações

Antes de explorarmos o resto dos comandos, é importante vermos como é que implementamos operações sobre valores. Um dos exemplos é o não-terminal "Expression"visto acima.
Primeiro temos de definir o não-terminal "Value"(e o seu correspondente para *floats* "ValueF").

```
def p_Value_ID(self, p):
    "Value : VAR"
    p[0] = f"pushg {self.fp[p[1]][0]}\n"

def p_Value_INT(self, p):
    "Value : INT"
    p[0] = f"pushi {p[1]}\n"

def p_Value_str(self, p):
    "Value : INTKW '(' String ')'"
    p[0] = f"{p[3]}atoi\n"

def p_Value_float(self, p):
    "Value : INTKW '(' ExpressionF ')'"
    p[0] = f"{p[3]}ftoi\n"

def p_ValueF_FLOAT(self, p):
    "ValueF : FLOAT"
    p[0] = f"pushf {p[1]}\n"

def p_ValueF_IDF(self, p):
    "ValueF : VARF"
    p[0] = f"pushg {self.fp[p[1]][0]}\n"

def p_ValueF_str(self, p):
    "ValueF : FLOATKW '(' String ')'"
    p[0] = f"{p[3]}atof\n"

def p_ValueF_int(self, p):
    "ValueF : FLOATKW '(' Expression ')'"
    p[0] = f"{p[3]}itof\n"

def p_Value_ArrayElem(self, p):
```

```
    "Value : VARA '[' Expression ']'"
    p[0] = f"pushgp\npushi {self.fp[p[1]][0]}\npadd\n{p[3]}loadn\n"

def p_Value_Array2dElem(self, p):
    "Value : VARA '[' Expression ']' '[' Expression ']'"
    p[0] = f"pushgp\npushi {self.fp[p[1]][0]}\npadd\n{p[3]}pushi
     ↪  {self.fp[p[1]][2][1]}\nmul\n{p[6]}add\nloadn\n"
```

Como podemos ver, podemos formar um Value a partir de uma variável, de um valor literal, de uma String ou de uma expressão em vírgula flutuante convertida para valor inteiro. O mesmo aplica-se para as regras do não-terminal ValueF.

Para ir buscar o valor de um elemento de um array precisamos de algo um pouco mais complexo. Primeiro precisamos do endereço base do array. Para isso, adicionamos ao endereço base da stack a posição do array na mesma. Depois, se for um array uni-dimensional, apenas precisamos de adicionar ao endereço base do array o índice especificado e usar a função `loadn` da VM para ir buscar o valor pretendido à stack. Para um array bidimensional precisamos primeiro de chegar à linha pretendida, o que requer que multipliquemos o número de colunas do array com a linha especificada. Por fim, somamos o número da coluna especificado, e temos assim a posição do elemento que queremos na stack da VM.

Podemos usar estes valores em operações aritméticas e lógicas, mas primeiro iremos reduzi-los para o não-terminal Expression/ExpressionF.

```
def p_Expression_Value(self, p):
    "Expression : Value"
    p[0] = p[1]

def p_ExpressionF_ValueF(self, p):
    "ExpressionF : ValueF"
    p[0] = p[1]
```

A partir deste não-terminal iremos implementar as operações básicas.

```
def p_Expression_plus(self, p):
    "Expression : Expression '+' Expression"
    p[0] =  p[1] + p[3] + "add\n"

def p_Expression_minus(self, p):
    "Expression : Expression '-' Expression"
    p[0] =  p[1] + p[3] + "sub\n"

def p_Expression_multiply(self, p):
    "Expression : Expression '*' Expression"
    p[0] =  p[1] + p[3] + "mul\n"

def p_Expression_divide(self, p):
    "Expression : Expression '/' Expression"
```

```
    p[0] =  p[1] + p[3] + "div\n"

def p_Expression_mod(self, p):
    "Expression : Expression '%' Expression"
    p[0] =  p[1] + p[3] + "mod\n"
```

Ao contrário do que foi feito nas aulas práticas da UC, no nosso trabalho prático definimos manualmente as precedências, com a seguinte estrutura de dados:

```
precedence = (
    ('left','OR'),
    ('left','AND'),
    ('left','NOT'),
    ('left','<','>','GE','LE','EQ','NE'),
    ('left','+','-'),
    ('left','*','/','\%'),
    ('right','UMINUS'),
    ('left','POW')
)
```

Nesta estrutura, suportada oficialmente pelo PLY, os valores mais abaixo têm maior precedência do que os valores acima, e igual precedência aos valores na mesma linha. Podemos ver que, por exemplo, as operações de multiplicação/divisão/módulo irão ser reduzidas antes das operações de adição/subtração.

Os valores "left"e "right"indicam se se reduz primeiro à esquerda ou à direita do sinal, ou seja, a sua associatividade. Nas operações aritméticas primeiro faz-se a operação da esquerda e só depois a da direita, se a prioridade dos dois lados for a mesma. Por outro lado, quando colocamos um '-' antes de um valor (neste caso o terminal UMINUS), o valor à direita é reduzido primeiro, logo este token tem associatividade à direita.

O token '-' já está definido como um literal, por isso não podemos definir UMINUS da forma convencional, mas o PLY permite-nos criar um token "artificial"da seguinte forma:

```
def p_Expression_uminus(self, p):
    "Expression : '-' Expression %prec UMINUS"
    p[0] = p[2] + "dup 1\ndup 1\nsub\nswap\nsub\n"

def p_Expression_uminus_f(self, p):
    "ExpressionF : '-' ExpressionF %prec UMINUS"
    p[0] = p[2] + "dup 1\ndup 1\nfsub\nswap\nfsub\n"
```

O `%prec UMINUS` diz ao Parser que esta regra deve ter a precedência atribuída ao token UMINUS na estrutura anterior, apesar desse token oficialmente não existir.

O facto de termos valores inteiros e em vírgula flutuante leva a que tenhamos que definir muitas das nossas regras duas vezes, visto que na VM as instruções para valores inteiros e flutuantes são diferentes. Por exemplo, para suportar as operações aritméticas de há um bocado com valores em vírgula flutuante, precisamos das 3 regras seguintes:

```
def p_ExpressionF_ops1(self, p):
    """ExpressionF : ExpressionF '+' ExpressionF
                   | ExpressionF '-' ExpressionF
                   | ExpressionF '*' ExpressionF
                   | ExpressionF '/' ExpressionF"""
    p[0] =  p[1] + p[3]
    if p[2] == '+': p[0] += "fadd\n"
    elif p[2] == '-': p[0] += "fsub\n"
    elif p[2] == '*': p[0] += "fmul\n"
    elif p[2] == '/': p[0] += "fdiv\n"

def p_ExpressionF_ops2(self, p):
    """ExpressionF : Expression '+' ExpressionF
                   | Expression '-' ExpressionF
                   | Expression '*' ExpressionF
                   | Expression '/' ExpressionF"""
    p[0] =  p[1] + "itof\n" + p[3]
    if p[2] == '+': p[0] += "fadd\n"
    elif p[2] == '-': p[0] += "fsub\n"
    elif p[2] == '*': p[0] += "fmul\n"
    elif p[2] == '/': p[0] += "fdiv\n"

def p_ExpressionF_ops3(self, p):
    """ExpressionF : ExpressionF '+' Expression
                   | ExpressionF '-' Expression
                   | ExpressionF '*' Expression
                   | ExpressionF '/' Expression"""
    p[0] =  p[1] + p[3] + "itof\n"
    if p[2] == '+': p[0] += "fadd\n"
    elif p[2] == '-': p[0] += "fsub\n"
    elif p[2] == '*': p[0] += "fmul\n"
    elif p[2] == '/': p[0] += "fdiv\n"
```

Infelizmente, isto é uma limitação da VM, todos os valores de uma operação devem ser do mesmo tipo, por isso se quisermos implementar floats de forma total precisamos destas regras todas.
Para dar maior precedência a operações entre parênteses apenas precisamos da seguinte regra:

```
def p_Expression_paren(self, p):
    """Expression : '(' Expression ')'
       ExpressionF : '(' ExpressionF ')'"""
    p[0] = p[2]
```

Temos ainda valores lógicos, que na verdade são valores inteiros, visto que a VM não suporta booleanos, por isso fazer a distinção não é necessário, mas decidimos fazê-la por uma questão de organização.

```
def p_Logical_Comparisons(self, p):
    """Logical : Expression '>' Expression
               | Expression '<' Expression
               | Expression GE Expression
               | Expression LE Expression
               | Expression EQ Expression
               | Expression NE Expression"""
    if p[2] == '>':
        p[0] = p[1] + p[3] + "sup\n"
    elif p[2] == '<':
        p[0] = p[1] + p[3] + "inf\n"
    elif p[2] == '>=':
        p[0] = p[1] + p[3] + "supeq\n"
    elif p[2] == '<=':
        p[0] = p[1] + p[3] + "infeq\n"
    elif p[2] == '==':
        p[0] = p[1] + p[3] + "equal\n"
    elif p[2] == '!=':
        p[0] = p[1] + p[3] + "equal\nnot\n"

def p_Logical_Comparisons_f(self, p):
    """Logical : ExpressionF '>' ExpressionF
               | ExpressionF '<' ExpressionF
               | ExpressionF GE ExpressionF
               | ExpressionF LE ExpressionF
               | ExpressionF EQ ExpressionF
               | ExpressionF NE ExpressionF"""
    if p[2] == '>':
        p[0] = p[1] + p[3] + "fsup\nftoi\n"
    elif p[2] == '<':
        p[0] = p[1] + p[3] + "finf\nftoi\n"
    elif p[2] == '>=':
        p[0] = p[1] + p[3] + "fsupeq\nftoi\n"
    elif p[2] == '<=':
        p[0] = p[1] + p[3] + "finfeq\nftoi\n"
    elif p[2] == '==':
        p[0] = p[1] + p[3] + "equal\n"
    elif p[2] == '!=':
        p[0] = p[1] + p[3] + "equal\nnot\n"

def p_Logical_Comparisons_f2(self, p):
    """Logical : ExpressionF '>' Expression
               | ExpressionF '<' Expression
               | ExpressionF GE Expression
               | ExpressionF LE Expression
               | ExpressionF EQ Expression
               | ExpressionF NE Expression"""
```

```
    p[0] = p[1] + p[3] + "itof\n"
    if p[2] == '>':
        p[0] += "fsup\nftoi\n"
    elif p[2] == '<':
        p[0] += "finf\nftoi\n"
    elif p[2] == '>=':
        p[0] += "fsupeq\nftoi\n"
    elif p[2] == '<=':
        p[0] += "finfeq\nftoi\n"
    elif p[2] == '==':
        p[0] += p[1] + p[3] + "equal\n"
    elif p[2] == '!=':
        p[0] += "equal\nnot\n"

def p_Logical_Comparisons_f3(self, p):
    """Logical : Expression '>' ExpressionF
               | Expression '<' ExpressionF
               | Expression GE ExpressionF
               | Expression LE ExpressionF
               | Expression EQ ExpressionF
               | Expression NE ExpressionF"""
    p[0] = p[1] + "itof\n" + p[3]
    if p[2] == '>':
        p[0] += "fsup\nftoi\n"
    elif p[2] == '<':
        p[0] += "finf\nftoi\n"
    elif p[2] == '>=':
        p[0] += "fsupeq\nftoi\n"
    elif p[2] == '<=':
        p[0] += "finfeq\nftoi\n"
    elif p[2] == '==':
        p[0] += p[1] + p[3] + "equal\n"
    elif p[2] == '!=':
        p[0] += "equal\nnot\n"


def p_Logical_Parens(self, p):
    "Logical : '(' Logical ')'"
    p[0] = p[2]

def p_Logical_AND(self, p):
    "Logical : Logical AND Logical"
    p[0] = p[1] + p[3] + "mul\n"

def p_Logical_OR(self, p):
    "Logical : Logical OR Logical"
    p[0] = p[1] + p[3] + "add\n"
```

```
def p_Logical_NOT(self, p):
    "Logical : NOT Logical"
    p[0] = p[2] + "not\n"
```

O último tipo que definimos é o tipo String, com um não-terminal homónimo, cujas regras no Parser são as seguintes:

```
def p_String(self, p):
    "String : TEXT"
    p[0] = "pushs " + '"' + p[1] + '"' + "\n"

def p_String_var(self, p):
    "String : VARS"
    p[0] = f"pushg {self.fp[p[1]][0]}\n"
```

Estas definições são muito semelhantes às dos não-terminais Value e ValueF. A única diferença notável é que colocamos as aspas manualmente na String, em vez de as manter no Lexer. Isto permite-nos usar aspas ou plicas na nossa linguagem de programação de forma intercambiável. Para referência, o token TEXT está definido da seguinte forma:

```
t_TEXT = r"'(\\'|[^'])*'|\"(\\\"|[^\"])*\""
```

Mudança de Valor de Variáveis

Tal como podemos declarar variáveis, também é possível alterar o seu valor, como seria de esperar. Tal como anteriormente, apenas iremos mostrar alguns exemplos.

```
def p_Redefine_f(self, p):
    "Redefine : VARF '=' ExpressionF"
    p[0] = f"{p[3]}storeg {self.fp[p[1]][0]}\n"

def p_Redefine_cast_2(self, p):
    "Redefine : VAR '=' ExpressionF"
    print(f"Warning in line {p.lineno(2)}: implicit casting of floating point value to
    ↪  integer.")
    p[0] = f"{p[3]}ftoi\nstoreg {self.fp[p[1]][0]}\n"
```

Vemos aqui em ação aquilo que implementámos no Lexer, isto é, apenas podemos dar um valor em vírgula flutuante (ExpressionF) a uma variável em vírgula flutuante (ou a uma variável inteira com *casting*). Fazer esta deteção de tipos no Parser iria ser muito mais complexo e desnecessário, pois desta forma temos uma solução simples que funciona.
Vemos também que recorremos ao nosso dicionário de variáveis para saber em que posição da stack da VM é que a variável se encontra.

```
def p_Redefine_Array2dElem(self, p):
    "Redefine : VARA '[' Expression ']' '[' Expression ']' '=' Expression"
    p[0] = f"pushgp\npushi {self.fp[p[1]][0]}\npadd\n{p[3]}pushi
    ↪  {self.fp[p[1]][2][1]}\nmul\n{p[6]}add\n{p[9]}storen\n"

def p_Redefine_Array2dElem_f(self, p):
    "Redefine : VARA '[' Expression ']' '[' Expression ']' '=' ExpressionF"
    p[0] = f"pushgp\npushi {self.fp[p[1]][0]}\npadd\n{p[3]}pushi
    ↪  {self.fp[p[1]][2][1]}\nmul\n{p[6]}add\n{p[9]}ftoi\nstoren\n"
```

A lógica por detrás destas regras para obter os endereços do array é a mesma que usámos na subsecção anterior para obter o elemento na posição pretendida, por isso não vale a pena voltar a explicar em detalhe.

```
def p_Redefine_ppmm(self, p):
    """Redefine : VAR PP
                | VAR MM"""
    i = self.fp[p[1]][0]
    p[0] = f"pushg {i}\npushi 1\n{'add' if p[2] == '++' else 'sub'}\nstoreg {i}\n"
```

Esta última regra diz respeito às operações de `x++` e `x--`, que são apenas uma forma mais rápida de escrever `x = x + 1` ou `x = x - 1`.

Input/Output

Lidar com input/output é fácil, visto que a VM trata da parte mais complicada, apenas temos de lhe fornecer os comandos certos.
Para imprimir valores no terminal temos as seguintes regras:

```
def p_Command_PRINT(self, p):
    """Command : PRINT '(' Expression ')' ';'
               | PRINT '(' Logical ')' ';'"""
    p[0] = p[3] + "writei\n"

def p_Command_PRINT_f(self, p):
    "Command : PRINT '(' ExpressionF ')' ';'"
    p[0] = p[3] + "writef\n"

def p_Command_PRINT_s(self, p):
    "Command : PRINT '(' String ')' ';'"
    p[0] = p[3] + "writes\n"

def p_Command_PRINTLN(self, p):
    """Command : PRINTLN '(' Expression ')' ';'
               | PRINTLN '(' Logical ')' ';'"""
```

```
    p[0] = p[3] + "writei\n" + "pushs \"\\n\"\n" + "writes\n"

def p_Command_PRINTLN_f(self, p):
    "Command : PRINTLN '(' ExpressionF ')' ';'"
    p[0] = p[3] + "writef\n" + "pushs \"\\n\"\n" + "writes\n"

def p_Command_PRINTLN_s(self, p):
    "Command : PRINTLN '(' String ')' ';'"
    p[0] = p[3] + "writes\n" + "pushs \"\\n\"\n" + "writes\n"
```

Tal como explicamos na especificação da linguagem, o comando `println` é igual ao comando `print` mas imprime também um caracter de *newline*. Podemos aqui ver que a sua compilação também acaba por ser igual, apenas diferindo nesse aspeto. Temos de ter regras diferentes para cada tipo de dados pois os comandos da VM para cada tipo são diferentes, como também é possível ver.

O input é ainda mais simples, visto que apenas temos de ler um valor do terminal.

```
def p_String_input(self, p):
    "String : INPUT '(' String ')'"
    p[0] = f"{p[3]}writes\nread\n"

def p_String_input_empty(self, p):
    "String : INPUT '(' ')'"
    p[0] = f"read\n"
```

Os valores lidos são reduzidos para uma String, que pode depois ser reduzida para um Value ou um ValueF, como vimos acima.

Instruções Condicionais

Para compilar instruções condicionais tivemos que recorrer a *jumps*, visto que é preciso saltar entre partes do programa.

```
def p_Command_if(self, p):
    "Command : IF Boolean '{' Commands '}'"
    p[0] = p[2] + f"jz l{self.labels_if}\n" + p[4] + f"l{self.labels_if}:\n"
    self.labels_if += 1

def p_Command_if_else(self, p):
    "Command : IF Boolean '{' Commands '}' Else"
    p[0] = p[2] + f"jz l{self.labels_if}\n" + p[4] + f"jump le{self.labels_if_else}\n" +
    ↪  f"l{self.labels_if}:\n" + p[6]
    self.labels_if += 1
    self.labels_if_else += 1

def p_Else(self, p):
```

```
    "Else : ELSE '{' Commands '}'"
    p[0] = p[3] + f"le{self.labels_if_else}:\n"

def p_Else_if(self, p):
    "Else : ELSE IF Boolean '{' Commands '}'"
    p[0] = p[3] + f"jz l{self.labels_if}\n" + p[5] + f"l{self.labels_if}:\n" +
    ↪  f"le{self.labels_if_else}:\n"
    self.labels_if += 1

def p_Else_if_else(self, p):
    "Else : ELSE IF Boolean '{' Commands '}' Else"
    p[0] = p[3] + f"jz l{self.labels_if}\n" + p[5] + f"jump le{self.labels_if_else}\n" +
    ↪  f"l{self.labels_if}:\n" + p[7]
    self.labels_if += 1
```

A lógica por detrás destas regras é a seguinte: primeiro coloca-se o resultado da condição no topo da stack (o não-terminal Boolean é apenas uma redução de Expression ou de Logical); depois verifica-se se esse valor é nulo, ou seja, se a condição é falsa; em caso afirmativo, salta-se para o fim da condição ou para a condição no *else*, se a regra contiver um *else*; em caso negativo, são executados os comandos dentro da instrução e, caso haja um ou mais *elses*, faz-se um salto para o fim da instrução. Este último salto garante que apenas é executado o código correspondente a uma condição. Se não o fizéssemos, no caso de um programa do género:

```
int x = 20;
if (x > 10) {println("Maior que 10"); }
else if (x > 5) { println("Maior que 5"); }
```

iriam ser executados ambos os `println`, quando apenas queremos que o primeiro seja executado.
Para identificar as *labels* dos saltos, temos contadores de *labels*, que contam quantas vezes uma dada *label* foi usada, e como esse valor faz parte do nome da *label*, nunca iremos ter *labels* repetidas, mesmo que tenhamos instruções condicionais aninhadas ou seguidas. Os contadores são inicializados a 0 ao correr o Parser.

Ciclos

A última funcionalidade que implementámos foi o parsing de ciclos.

```
def p_Command_for(self, p):
    """Command : FOR '(' Redefine ';' Boolean ';' Redefine ')' '{' Commands '}'
               | FOR '(' Empty ';' Boolean ';' Redefine ')' '{' Commands '}'"""
    p[0] = (p[3] if p[3] else "") + f"lc{self.labels_cycle}:\n" + p[5] + f"jz
    ↪  lb{self.labels_cycle_exit}\n" + p[10] + p[7] + f"jump lc{self.labels_cycle}\n" +
    ↪  f"lb{self.labels_cycle_exit}:\n"
    self.labels_cycle += 1
    self.labels_cycle_exit += 1
```

```
def p_Command_while(self, p):
    "Command : WHILE Boolean '{' Commands '}'"
    p[0] = f"lc{self.labels_cycle}:\n" + p[2] + f"jz lb{self.labels_cycle_exit}\n" + p[4]
    ↪  + f"jump lc{self.labels_cycle}\n" + f"lb{self.labels_cycle_exit}:\n"
    self.labels_cycle += 1
    self.labels_cycle_exit += 1
```

Tal como nas instruções condicionais, temos contadores que são incrementados cada vez que o tradutor faz o parsing de um ciclo. A maior diferença entre estes dois tipos de instruções é que aqui, no fim de executar as instruções, voltamos à condição, em vez de sair da instrução. Apenas saímos do ciclo quando a condição for falsa. Entre o ciclo "for"e o ciclo "while", a única diferença no parsing é que para o ciclo "for"temos de incluir a operação de incrementação da variável no conjunto de comandos a executar e iniciar a execução puxando a variável especificada para o topo da stack.

Extra: exponenciação

A partir das nossas regras para ciclos, decidimos implementar uma funcionalidade presente em quase todas as linguagens de programação: exponenciação. Dentro da nossa linguagem, basta escrever, por exemplo, $3 \wedge 2$ ou `3**2`, e o nosso parser converte esta instrução para um ciclo que calculará o valor desta operação na VM. A regra usada para implementar esta funcionalidade é a seguinte:

```
    def p_Expression_pow(self, p):
        "Expression : Expression POW Expression"
        p[0] = f"""pushi 1
{p[3]}pow{self.labels_pow}:
pushsp
load -1
pushi 0
sup
jz endpow{self.labels_pow}
pushsp
dup 1
load -2
{p[1]}mul
store -2
pushsp
dup 1
load -1
pushi 1
sub
store -1
jump pow{self.labels_pow}
endpow{self.labels_pow}:
pop 1
"""
```

4.3 Main

O ficheiro `main.py` do projeto também tem algumas caraterísticas a notar.
Para o correr, podemos executar o seguinte comando:

```
> python main.py [nome do ficheiro de input] [nome do ficheiro de output]
```

Se não lhe forem dados argumentos, o programa corre em modo interativo, onde lê código introduzido pelo utilizador diretamente a partir do terminal e apresenta o código respetivo na linguagem Assembly da VM.
Caso o utilizador tenha fornecido um ficheiro de input, é lido o programa nesse ficheiro e convertido para código-máquina. O ficheiro de output terá o nome do ficheiro de input com a extensão .vm se não for fornecido um nome.
É possível ainda correr o programa com a flag -h"ou --help". Desta forma, será apresentada uma mensagem de ajuda a explicar como correr o programa.

Capítulo 5

Conclusão

Este projeto permitiu-nos aprofundar os conhecimentos adquiridos nas aulas de Processamento de Linguagens de uma forma prática e interessante. Passamos a ter uma ideia de como é que os compiladores como o `gcc` funcionam, algo que nos poderá vir a ser muito útil no nosso futuro académico e profissional. Para além disso, apesar de já termos usado linguagem Assembly antes, usar a VM deu-nos a oportunidade de relembrar como é que as linguagens máquina funcionam, especialmente a parte dos jumps e da stack.

Gostaríamos de ter implementado mais funcionalidades, como a possibilidade de definir sub-programas na linguagem de programação, mas não fomos capazes de o fazer por restrições temporais. Porém, fizemos tudo o que nos foi pedido no enunciado, por isso estamos bastante satisfeitas com o nosso projeto final.

Temos ainda no Anexo A alguns exemplos de execução do nosso projeto para alguns programas-fonte escritos na nossa linguagem de programação.

Apêndice A

Exemplos de Utilização

A.1 Exemplo 1 - Quadrado ou Não Quadrado, Eis a Questão

Código LGBTI:

```
int n1 = 0, n2 = 0, n3 = 0, n4 = 0;
n1 = int(input("Introduza o valor do 1º lado: "));
n2 = int(input("Introduza o valor do 2º lado: "));
n3 = int(input("Introduza o valor do 3º lado: "));
n4 = int(input("Introduza o valor do 4º lado: "));

if n1 == n2 and n2 == n3 and n3 == n4 {
    println("Estes lados formam um quadrado.");
}
else {
    println("Estes lados não podem formar um quadrado.");
}
```

Código máquina gerado:

```
pushn 4
start
pushi 0
storeg 0
pushi 0
storeg 1
pushi 0
storeg 2
pushi 0
storeg 3
pushs "Introduza o valor do 1º lado: "
writes
read
atoi
```

```
storeg 0
pushs "Introduza o valor do 2º lado: "
writes
read
atoi
storeg 1
pushs "Introduza o valor do 3º lado: "
writes
read
atoi
storeg 2
pushs "Introduza o valor do 4º lado: "
writes
read
atoi
storeg 3
pushg 0
pushg 1
equal
pushg 1
pushg 2
equal
mul
pushg 2
pushg 3
equal
mul
jz l0
pushs "Estes lados formam um quadrado."
writes
pushs "\n"
writes
jump le0
l0:
pushs "Estes lados não podem formar um quadrado."
writes
pushs "\n"
writes
le0:
stop
```

A.2 Exemplo 2 - Ler N, Ler N números, e Escrever o Menor Deles

Código LGBTI:

```
int n = int(input("Quantos valores pretende ler? "));
int i, v, smallest;
if n < 1 { println ("Error - 'n' must be a natural number!"); }
else {
    for(i = 0; i < n; i++) {
        print("Valor ");
        print(i);
        v = int(input(": "));
        if i == 1 { smallest = v; }
        else if v < smallest {
            smallest = v;
        }
    }
    print("O menor valor lido é ");
    println(smallest);
}
```

Código máquina gerado:

```
pushs "Quantos valores pretende ler? "
writes
read
atoi
pushi 0
pushi 0
pushi 0
start
pushg 0
pushi 1
inf
jz l2
pushs "Error - 'n' must be a natural number!"
writes
pushs "\n"
writes
jump le1
l2:
pushi 0
storeg 1
lc0:
pushg 1
pushg 0
inf
jz lb0
pushs "Valor "
writes
```

```
pushg 1
writei
pushs ": "
writes
read
atoi
storeg 2
pushg 1
pushi 1
equal
jz l1
pushg 2
storeg 3
jump le0
l1:
pushg 2
pushg 3
inf
jz l0
pushg 2
storeg 3
l0:
le0:
pushg 1
pushi 1
add
storeg 1
jump lc0
lb0:
pushs "O menor valor lido é "
writes
pushg 3
writei
pushs "\n"
writes
le1:
stop
```

A.3 Exemplo 3 - Ler N Números e Imprimir o Produtório

Código LGBTI:

```
int N = 5;
int i, v, prod = 1;
print("Introduza ");
print(N);
```

```
println(" valores, um por linha: ");
while i < N {
    v = int(input());
    prod = prod * v;
    i++;
}
print("O produtório destes valores é ");
println(prod);
```

Código máquina gerado:

```
pushi 5 // 0
pushi 0 // 1
pushi 0 // 2
pushi 1 // 3
start
pushs "Introduza "
writes
pushg 0
writei
pushs " valores, um por linha: "
writes
pushs "\n"
writes
lc0:
pushg 1
pushg 0
inf
jz lb0
read
atoi
storeg 2
pushg 3
pushg 2
mul
storeg 3
pushg 1
pushi 1
add
storeg 1
jump lc0
lb0:
pushs "O produtório destes valores é "
writes
pushg 3
writei
pushs "\n"
```

```
writes
stop
```

A.4 Exemplo 4 - Contar e Imprimir os Números Ímpares de uma Sequência de Números Naturais

Código LGBTI:

```
int cont = 0;
int num = -1;

println("Indique os números naturais (insira 0 para parar):");
while(num != 0) {
    num = int(input(""));
    if num > 0 && num % 2 != 0 {
        cont++;
        println("Este número é ímpar");
    }
}

print("Total: ");
print(cont);
println(" números ímpares.");
```

Código máquina gerado:

```
pushi 0
pushi 1
dup 1
dup 1
sub
swap
sub
start
pushs "Indique os números naturais (insira 0 para parar):"
writes
pushs "\n"
writes
lc0:
pushg 1
pushi 0
equal
not
jz lb0
pushs ""
```

```
writes
read
atoi
storeg 1
pushg 1
pushi 0
sup
pushg 1
pushi 2
mod
pushi 0
equal
not
mul
jz l0
pushg 0
pushi 1
add
storeg 0
pushs "Este número é ímpar"
writes
pushs "\n"
writes
l0:
jump lc0
lb0:
pushs "Total: "
writes
pushg 0
writei
pushs " números ímpares."
writes
pushs "\n"
writes
stop
```

A.5 Exemplo 5 - Ler e Armazenar N Números num Array, e Imprimir pela Ordem Inversa

Código LGBTI:

```
int N = 5;
int a[5];
int i;
for(i = 0; i < N; i++) {
    a[i] = int(input("Introduza um valor: "));
```

```
}

for(i = N - 1; i >= 0; i--) {
    print(a[i]);
    if i != 0 { print(","); }
}
```

Código máquina gerado:

```
pushi 5
pushn 5
pushi 0
start
pushi 0
storeg 6
lc0:
pushg 6
pushg 0
inf
jz lb0
pushgp
pushi 1
padd
pushg 6
pushs "Introduza um valor: "
writes
read
atoi
storen
pushg 6
pushi 1
add
storeg 6
jump lc0
lb0:
pushg 0
pushi 1
sub
storeg 6
lc1:
pushg 6
pushi 0
supeq
jz lb1
pushgp
pushi 1
padd
```

```
pushg 6
loadn
writei
pushg 6
pushi 0
equal
not
jz l0
pushs ","
writes
l0:
pushg 6
pushi 1
sub
storeg 6
jump lc1
lb1:
stop
```

A.6 Exemplo Extra - Soma de Matrizes

Código LGBTI:

```
int N = 3;
int a[][] = [[1,2,3],[4,5,6],[7,8,9]];
int b[][] = [[9,8,7],[6,5,4],[3,2,1]];
int c[3][3];
int i, j;

for(i = 0; i < N; i++) {
    for(j = 0; j < N; j++) {
        c[i][j] = a[i][j] + b[i][j];
    }
}

println("[");
for(i = 0; i < N; i++) {
    print("[");
    for(j = 0; j < N; j++) {
        print(c[i][j]);
        if j != N - 1 { print(","); }
    }
    println("]");
}
println("]");
```

Código máquina gerado:

```
pushi 3
pushi 1
pushi 2
pushi 3
pushi 4
pushi 5
pushi 6
pushi 7
pushi 8
pushi 9
pushi 9
pushi 8
pushi 7
pushi 6
pushi 5
pushi 4
pushi 3
pushi 2
pushi 1
pushn 9
pushi 0
pushi 0
start
pushi 0
storeg 28
lc1:
pushg 28
pushg 0
inf
jz lb1
pushi 0
storeg 29
lc0:
pushg 29
pushg 0
inf
jz lb0
pushgp
pushi 19
padd
pushg 28
pushi 3
mul
pushg 29
add
```

```
pushgp
pushi 1
padd
pushg 28
pushi 3
mul
pushg 29
add
loadn
pushgp
pushi 10
padd
pushg 28
pushi 3
mul
pushg 29
add
loadn
add
storen
pushg 29
pushi 1
add
storeg 29
jump lc0
lb0:
pushg 28
pushi 1
add
storeg 28
jump lc1
lb1:
pushs "["
writes
pushs "\n"
writes
pushi 0
storeg 28
lc3:
pushg 28
pushg 0
inf
jz lb3
pushs "["
writes
pushi 0
storeg 29
lc2:
```

```
pushg 29
pushg 0
inf
jz lb2
pushgp
pushi 19
padd
pushg 28
pushi 3
mul
pushg 29
add
loadn
writei
pushg 29
pushg 0
pushi 1
sub
equal
not
jz l0
pushs ","
writes
l0:
pushg 29
pushi 1
add
storeg 29
jump lc2
lb2:
pushs "]"
writes
pushs "\n"
writes
pushg 28
pushi 1
add
storeg 28
jump lc3
lb3:
pushs "]"
writes
pushs "\n"
writes
stop
```

Apêndice B

Main

```
#!/usr/bin/env python3

# Linguagem Genérica Baseada nas Típicas Imperativas

from myParser import Parser
import sys

parser = Parser()

parser.build()

if len(sys.argv) < 2:
    f = ""

    while line := input():
        f += line + "\n"

    result = parser.parser.parse(f)
    print(result)
elif sys.argv[1] == "-h" or sys.argv[1] == "--help":
    print("""Compilador da Linguagem Genérica Baseada nas Típicas Imperativas

Uso do comando: python main.py [nome do ficheiro de input] [nome do ficheiro de output]

O nome do ficheiro de output será igual ao do ficheiro de input, com a extensão '.vm',
↪  caso não seja fornecido um nome.

Se não for fornecido um ficheiro de input, o programa irá correr em modo "interativo", no
↪  qual é possível introduzir um pedaço de código no terminal e será apresentada a sua
↪  versão "compilada", em código-máquina.""")
else:
    with open(sys.argv[1],"r") as f:
        result = parser.parser.parse(f.read())
```

```
    if result:
        with open(sys.argv[1].strip(".\\").split('.')[0] + '.vm' if len(sys.argv) < 3
        ↪  else sys.argv[2], "w", newline='\n') as r:
            r.write(result)
```

Apêndice C

Lexer

```
import ply.lex as lex

INT = 0
FLOAT = 1
STRING = 2
ARRAY = 3

class Lexer:

    def __init__(self, fp : dict):
        self.fp = fp

    reserved = {
        'int': 'INTKW',
        'float': 'FLOATKW',
        'str': 'STRKW',
        'print': "PRINT",
        'println': "PRINTLN",
        'or': 'OR',
        'and': 'AND',
        'not': 'NOT',
        'if': 'IF',
        'else': 'ELSE',
        'input': 'INPUT',
        'for': 'FOR',
        'while': 'WHILE'
    }

    tokens = [
        'INT',
        'FLOAT',
        'ID',
        'VAR',
```

```
    'VARF',
    'VARS',
    'VARA',
    'POW',
    'GE',
    'LE',
    'EQ',
    'NE',
    'TEXT',
    'PP',
    'MM'
] + list(reserved.values())

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

def t_FLOAT(self, t):
    r'(\d*)?\.\d+(e(?:\+|-)\d+)?'
    t.value = float(t.value)
    return t

def t_INT(self, t):
    r'\d+'
    t.value = int(t.value)
    return t

t_POW = r'\*\*|\^'
t_GE = r'>='
t_LE = r'<='
t_EQ = r'=='
t_NE = r'!='
t_AND = r'&&'
t_OR = r'\|\|'
t_NOT = r'!'
t_PP = r'\+\+'
t_MM = r'\-\-'

def t_ID(self, t):
    r'[a-zA-Z_][a-zA-Z0-9_\']*'
    t.type = Lexer.reserved.get(t.value,"ID")
    v = self.fp.get(t.value, None)
    if v is not None:
        vartype = v[1]
        if vartype == INT: t.type = "VAR"
        elif vartype == FLOAT: t.type = "VARF"
        elif vartype == STRING: t.type = "VARS"
        elif vartype == ARRAY: t.type = "VARA"
    return t
```

```
    t_TEXT = r"'(\\'|[^'])*'|\"(\\\"|[^\"])*\""

    t_ignore = ' \t'

    def t_newline(self, t):
        r'\n'
        t.lexer.lineno += 1

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

    def build(self, **kwargs):
        self.lexer = lex.lex(module=self, **kwargs)
```

Apêndice D

Parser

```
import ply.yacc as yacc

from myLexer import Lexer

INT = 0
FLOAT = 1
STRING = 2
ARRAY = 3
class Parser:

    tokens = Lexer.tokens

    def p_Program(self, p):
        "Program : Attribs Commands"
        p[0] = p[1] + "start\n" + p[2] + "stop\n"

    def p_Program_noAttribs(self, p):
        "Program : Commands"
        p[0] = "start\n" + p[1] + "stop\n"

    def p_Program_noCommands(self, p):
        "Program : Attribs"
        print("Error: no instructions found.")
        raise SyntaxError

    def p_Program_error(self, p):
        "Program : error"
        print("Aborting...")
        exit(1)

    def p_Attribs(self, p):
        "Attribs : Attribs Attrib"
        p[0] = p[1] + p[2]
```

```
def p_Attribs_single(self, p):
    "Attribs : Attrib"
    p[0] = p[1]

def p_Commands(self, p):
    "Commands : Commands Command"
    p[0] = p[1] + p[2]

def p_Commands_single(self, p):
    "Commands : Command"
    p[0] = p[1]

def p_Attrib_Array(self, p):
    "Attrib : INTKW ID '[' INT ']' ';'"
    if p[2] not in self.fp:
        self.fp[p[2]] = (self.stack_size,ARRAY,p[4])
        self.stack_size += p[4]
    else:
        print(f"Error in line {p.lineno(2)}: variable {p[2]} was previously
        ↪  declared.")
        raise SyntaxError
    p[0] = f"pushn {p[4]}\n"

def p_Attrib_Array_2d(self, p):
    "Attrib : INTKW ID '[' INT ']' '[' INT ']' ';'"
    if p[2] not in self.fp:
        self.fp[p[2]] = (self.stack_size,ARRAY,(p[4],p[7]))
        self.stack_size += p[4] * p[7]
    else:
        print(f"Error in line {p.lineno(2)}: variable {p[2]} was previously
        ↪  declared.")
        raise SyntaxError
    p[0] = f"pushn {p[4]*p[7]}\n"

def p_Array_2d(self, p):
    "Array2d : '[' Arrays ']'"
    p[0] = p[2]

def p_Arrays(self, p):
    "Arrays : Arrays ',' Array "
    if len(p[3]) != len(p[1][0]):
        print(f"Error in line {p.lineno(2)}: two dimensional array must have rows of
        ↪  equal length.")
        raise SyntaxError
    p[0] = p[1]
    p[0].append(p[3])
```

```
def p_Arrays_single(self, p):
    "Arrays : Array"
    p[0] = [p[1]]

def p_Array(self, p):
    "Array : '[' Elems ']' "
    p[0] = p[2]

def p_Elems(self, p):
    "Elems : Elems ',' INT"
    p[0] = p[1]
    p[0].append(p[3])

def p_Elems_single(self, p):
    "Elems : INT"
    p[0] = [p[1]]

def p_Attrib_Array_elems(self, p):
    "Attrib : INTKW ID '[' INT ']' '=' Array ';'"
    "Attrib : INTKW ID '[' Empty ']' '=' Array ';'"
    if not p[4]: p[4] = len(p[7])
    if p[2] not in self.fp:
        self.fp[p[2]] = (self.stack_size,ARRAY,p[4])
        self.stack_size += p[4]
    else:
        print(f"Error in line {p.lineno(2)}: variable {p[2]} was previously
        ↪  declared.")
        raise SyntaxError

    if len(p[7]) != p[4]:
        print(f"Error in line {p.lineno(2)}: number of array elements different from
        ↪  the defined value.")
        raise SyntaxError
    p[0] = ""
    for elem in p[7]:
        p[0] += f"pushi {elem}\n"

def p_Attrib_Array_2d_elems(self, p):
    """Attrib : INTKW ID '[' INT ']' '[' INT ']' '=' Array2d ';'
              | INTKW ID '[' INT ']' '[' Empty ']' '=' Array2d ';'
              | INTKW ID '[' Empty ']' '[' INT ']' '=' Array2d ';'
              | INTKW ID '[' Empty ']' '[' Empty ']' '=' Array2d ';'"""

    if not p[4]: p[4] = len(p[10])
    if not p[7]: p[7] = len(p[10][0])

    if len(p[10]) != p[4]:
```

```
            print(f"Error in line {p.lineno(2)}: mismatch between expected number of rows
            ↪  and actual number of rows found.")
            raise SyntaxError
        if len(p[10][0]) != p[7]:
            print(f"Error in line {p.lineno(2)}: mismatch between expected number of
            ↪  columns and actual number of columns found.")
            raise SyntaxError

        if p[2] not in self.fp:
            self.fp[p[2]] = (self.stack_size,ARRAY,(p[4],p[7]))
            self.stack_size += p[4] * p[7]
        else:
            print(f"Error in line {p.lineno(2)}: variable {p[2]} was previously
            ↪  declared.")
            raise SyntaxError

        p[0] = ""
        for row in p[10]:
            for elem in row:
                p[0] += f"pushi {elem}\n"

    def p_Attrib(self, p):
        "Attrib : INTKW AttribsInt ';'"
        p[0] = p[2]

    def p_AttribsInt(self, p):
        "AttribsInt : AttribsInt ',' AttribInt"
        p[0] = p[1] + p[3]

    def p_AttribsInt_single(self, p):
        "AttribsInt : AttribInt"
        p[0] = p[1]

    def p_AttribInt(self, p):
        """AttribInt : ID '=' Expression
                     | ID"""
        if p[1] not in self.fp:
            self.fp[p[1]] = (self.stack_size,INT)
            self.stack_size += 1
        else:
            print(f"Error in line {p.lineno(1)}: variable {p[1]} was previously
            ↪  declared.")
            raise SyntaxError
        p[0] = p[3] if len(p) > 2 else 'pushi 0\n'

    def p_Attrib_f(self, p):
        "Attrib : FLOATKW AttribsFloat ';'"
        p[0] = p[2]
```

```
def p_AttribsFloat(self, p):
    "AttribsFloat : AttribsFloat ',' AttribFloat"
    p[0] = p[1] + p[3]

def p_AttribsFloat_single(self, p):
    "AttribsFloat : AttribFloat"
    p[0] = p[1]

def p_AttribFloat(self, p):
    """AttribFloat : ID '=' ExpressionF
                   | ID"""
    if p[1] not in self.fp:
        self.fp[p[1]] = (self.stack_size,FLOAT)
        self.stack_size += 1
    else:
        print(f"Error in line {p.lineno(1)}: variable {p[1]} was previously
        ↪  declared.")
        raise SyntaxError
    p[0] = p[3] if len(p) > 2 else 'pushf 0.0\n'

def p_Attrib_s(self, p):
    "Attrib : STRKW AttribsString ';'"
    p[0] = p[2]

def p_AttribsString(self, p):
    "AttribsString : AttribsString ',' AttribString"
    p[0] = p[1] + p[3]

def p_AttribsString_single(self, p):
    "AttribsString : AttribString"
    p[0] = p[1]

def p_AttribString(self, p):
    "AttribString : ID '=' String ';'"
    if p[1] not in self.fp:
        self.fp[p[1]] = (self.stack_size,STRING)
        self.stack_size += 1
    else:
        print(f"Error in line {p.lineno(1)}: variable {p[1]} was previously
        ↪  declared.")
        raise SyntaxError
    p[0] = p[3]

def p_Attrib_Error(self, p):
    """Attrib : INTKW error ';'
              | FLOATKW error ';'
              | STRKW error ';'"""
```

```
    p[0] = ""

def p_Command_Redefine(self, p):
    "Command : Redefine ';'"
    p[0] = p[1]

def p_Redefine(self, p):
    "Redefine : VAR '=' Expression"
    p[0] = f"{p[3]}storeg {self.fp[p[1]][0]}\n"

def p_Redefine_ppmm(self, p):
    """Redefine : VAR PP
                | VAR MM"""
    i = self.fp[p[1]][0]
    p[0] = f"pushg {i}\npushi 1\n{'add' if p[2] == '++' else 'sub'}\nstoreg {i}\n"

def p_Redefine_ArrayElem(self, p):
    "Redefine : VARA '[' Expression ']' '=' Expression"
    p[0] = f"pushgp\npushi {self.fp[p[1]][0]}\npadd\n{p[3]}{p[6]}storen\n"

def p_Redefine_ArrayElem_f(self, p):
    "Redefine : VARA '[' Expression ']' '=' ExpressionF"
    p[0] = f"pushgp\npushi {self.fp[p[1]][0]}\npadd\n{p[3]}{p[6]}ftoi\nstoren\n"

def p_Redefine_Array2dElem(self, p):
    "Redefine : VARA '[' Expression ']' '[' Expression ']' '=' Expression"
    p[0] = f"pushgp\npushi {self.fp[p[1]][0]}\npadd\n{p[3]}pushi
    ↪  {self.fp[p[1]][2][1]}\nmul\n{p[6]}add\n{p[9]}storen\n"

def p_Redefine_Array2dElem_f(self, p):
    "Redefine : VARA '[' Expression ']' '[' Expression ']' '=' ExpressionF"
    p[0] = f"pushgp\npushi {self.fp[p[1]][0]}\npadd\n{p[3]}pushi
    ↪  {self.fp[p[1]][2][1]}\nmul\n{p[6]}add\n{p[9]}ftoi\nstoren\n"

def p_Redefine_pp_f(self, p):
    """Redefine : VARF PP
                | VARF MM"""
    i = self.fp[p[1]][0]
    p[0] = f"pushg {i}\npushf 1.0\n{'fadd' if p[2] == '++' else 'fsub'}\nstoreg
    ↪  {i}\n"

def p_Redefine_f(self, p):
    "Redefine : VARF '=' ExpressionF"
    p[0] = f"{p[3]}storeg {self.fp[p[1]][0]}\n"

def p_Redefine_s(self, p):
    "Redefine : VARS '=' String"
    p[0] = f"{p[3]}storeg {self.fp[p[1]][0]}\n"
```

```
def p_Redefine_cast(self, p):
    "Redefine : VARF '=' Expression"
    p[0] = f"{p[3]}itof\nstoreg {self.fp[p[1]][0]}\n"

def p_Redefine_cast_2(self, p):
    "Redefine : VAR '=' ExpressionF"
    print(f"Warning in line {p.lineno(2)}: implicit casting of floating point value
    ↪  to integer.")
    p[0] = f"{p[3]}ftoi\nstoreg {self.fp[p[1]][0]}\n"

def p_Command_PRINT(self, p):
    """Command : PRINT '(' Expression ')' ';'
               | PRINT '(' Logical ')' ';'"""
    p[0] = p[3] + "writei\n"

def p_Command_PRINT_f(self, p):
    "Command : PRINT '(' ExpressionF ')' ';'"
    p[0] = p[3] + "writef\n"

def p_Command_PRINT_s(self, p):
    "Command : PRINT '(' String ')' ';'"
    p[0] = p[3] + "writes\n"

def p_Command_PRINTLN(self, p):
    """Command : PRINTLN '(' Expression ')' ';'
               | PRINTLN '(' Logical ')' ';'"""
    p[0] = p[3] + "writei\n" + "pushs \"\\n\"\n" + "writes\n"

def p_Command_PRINTLN_f(self, p):
    "Command : PRINTLN '(' ExpressionF ')' ';'"
    p[0] = p[3] + "writef\n" + "pushs \"\\n\"\n" + "writes\n"

def p_Command_PRINTLN_s(self, p):
    "Command : PRINTLN '(' String ')' ';'"
    p[0] = p[3] + "writes\n" + "pushs \"\\n\"\n" + "writes\n"

def p_Command_if(self, p):
    "Command : IF Boolean '{' Commands '}'"
    p[0] = p[2] + f"jz l{self.labels_if}\n" + p[4] + f"l{self.labels_if}:\n"
    self.labels_if += 1

def p_Command_if_else(self, p):
    "Command : IF Boolean '{' Commands '}' Else"
    p[0] = p[2] + f"jz l{self.labels_if}\n" + p[4] + f"jump
    ↪  le{self.labels_if_else}\n" + f"l{self.labels_if}:\n" + p[6]
    self.labels_if += 1
    self.labels_if_else += 1
```

```
def p_Else(self, p):
    "Else : ELSE '{' Commands '}'"
    p[0] = p[3] + f"le{self.labels_if_else}:\n"

def p_Else_if(self, p):
    "Else : ELSE IF Boolean '{' Commands '}'"
    p[0] = p[3] + f"jz l{self.labels_if}\n" + p[5] + f"l{self.labels_if}:\n" +
    ↪  f"le{self.labels_if_else}:\n"
    self.labels_if += 1

def p_Else_if_else(self, p):
    "Else : ELSE IF Boolean '{' Commands '}' Else"
    p[0] = p[3] + f"jz l{self.labels_if}\n" + p[5] + f"jump
    ↪  le{self.labels_if_else}\n" + f"l{self.labels_if}:\n" + p[7]
    self.labels_if += 1

def p_Command_for(self, p):
    """Command : FOR '(' Redefine ';' Boolean ';' Redefine ')' '{' Commands '}'
               | FOR '(' Empty ';' Boolean ';' Redefine ')' '{' Commands '}'"""
    p[0] = (p[3] if p[3] else "") + f"lc{self.labels_cycle}:\n" + p[5] + f"jz
    ↪  lb{self.labels_cycle_exit}\n" + p[10] + p[7] + f"jump
    ↪  lc{self.labels_cycle}\n" + f"lb{self.labels_cycle_exit}:\n"
    self.labels_cycle += 1
    self.labels_cycle_exit += 1

def p_Command_while(self, p):
    "Command : WHILE Boolean '{' Commands '}'"
    p[0] = f"lc{self.labels_cycle}:\n" + p[2] + f"jz lb{self.labels_cycle_exit}\n" +
    ↪  p[4] + f"jump lc{self.labels_cycle}\n" + f"lb{self.labels_cycle_exit}:\n"
    self.labels_cycle += 1
    self.labels_cycle_exit += 1

precedence = (
    ('left','OR'),
    ('left','AND'),
    ('left','NOT'),
    ('left','<','>','GE','LE','EQ','NE'),
    ('left','+','-'),
    ('left','*','/','%'),
    ('right','UMINUS'),
    ('left','POW')
)

def p_Expression_plus(self, p):
    "Expression : Expression '+' Expression"
    p[0] =  p[1] + p[3] + "add\n"
```

```
def p_Expression_minus(self, p):
    "Expression : Expression '-' Expression"
    p[0] =  p[1] + p[3] + "sub\n"

def p_Expression_multiply(self, p):
    "Expression : Expression '*' Expression"
    p[0] =  p[1] + p[3] + "mul\n"

def p_Expression_divide(self, p):
    "Expression : Expression '/' Expression"
    p[0] =  p[1] + p[3] + "div\n"

def p_Expression_mod(self, p):
    "Expression : Expression '%' Expression"
    p[0] =  p[1] + p[3] + "mod\n"

def p_ExpressionF_ops1(self, p):
    """ExpressionF : ExpressionF '+' ExpressionF
                   | ExpressionF '-' ExpressionF
                   | ExpressionF '*' ExpressionF
                   | ExpressionF '/' ExpressionF"""
    p[0] =  p[1] + p[3]
    if p[2] == '+': p[0] += "fadd\n"
    elif p[2] == '-': p[0] += "fsub\n"
    elif p[2] == '*': p[0] += "fmul\n"
    elif p[2] == '/': p[0] += "fdiv\n"

def p_ExpressionF_ops2(self, p):
    """ExpressionF : Expression '+' ExpressionF
                   | Expression '-' ExpressionF
                   | Expression '*' ExpressionF
                   | Expression '/' ExpressionF"""
    p[0] =  p[1] + "itof\n" + p[3]
    if p[2] == '+': p[0] += "fadd\n"
    elif p[2] == '-': p[0] += "fsub\n"
    elif p[2] == '*': p[0] += "fmul\n"
    elif p[2] == '/': p[0] += "fdiv\n"

def p_ExpressionF_ops3(self, p):
    """ExpressionF : ExpressionF '+' Expression
                   | ExpressionF '-' Expression
                   | ExpressionF '*' Expression
                   | ExpressionF '/' Expression"""
    p[0] =  p[1] + p[3] + "itof\n"
    if p[2] == '+': p[0] += "fadd\n"
    elif p[2] == '-': p[0] += "fsub\n"
    elif p[2] == '*': p[0] += "fmul\n"
    elif p[2] == '/': p[0] += "fdiv\n"
```

```
    def p_Expression_pow(self, p):
        "Expression : Expression POW Expression"
        p[0] = f"""pushi 1
{p[3]}pow{self.labels_pow}:
pushsp
load -1
pushi 0
sup
jz endpow{self.labels_pow}
pushsp
dup 1
load -2
{p[1]}mul
store -2
pushsp
dup 1
load -1
pushi 1
sub
store -1
jump pow{self.labels_pow}
endpow{self.labels_pow}:
pop 1
"""
        self.labels_pow += 1

    def p_ExpressionF_pow(self, p):
        "ExpressionF : ExpressionF POW Expression"
        p[0] = f"""pushf 1.0
{p[3]}pow{self.labels_pow}:
pushsp
load -1
pushi 0
sup
jz endpow{self.labels_pow}
pushsp
dup 1
load -2
{p[1]}fmul
store -2
pushsp
dup 1
load -1
pushi 1
sub
store -1
jump pow{self.labels_pow}
```

```
endpow{self.labels_pow}:
pop 1
"""
        self.labels_pow += 1

    def p_Expression_uplus(self, p):
        """Expression : '+' Expression
           ExpressionF : '+' ExpressionF"""
        p[0] = p[2]

    def p_Expression_uminus(self, p):
        "Expression : '-' Expression %prec UMINUS"
        p[0] = p[2] + "dup 1\ndup 1\nsub\nswap\nsub\n"

    def p_Expression_uminus_f(self, p):
        "ExpressionF : '-' ExpressionF %prec UMINUS"
        p[0] = p[2] + "dup 1\ndup 1\nfsub\nswap\nfsub\n"

    def p_Expression_paren(self, p):
        """Expression : '(' Expression ')'
           ExpressionF : '(' ExpressionF ')'"""
        p[0] = p[2]

    def p_Expression_Value(self, p):
        "Expression : Value"
        p[0] = p[1]

    def p_ExpressionF_ValueF(self, p):
        "ExpressionF : ValueF"
        p[0] = p[1]

    def p_Value_ID(self, p):
        "Value : VAR"
        p[0] = f"pushg {self.fp[p[1]][0]}\n"

    def p_Value_INT(self, p):
        "Value : INT"
        p[0] = f"pushi {p[1]}\n"

    def p_Value_str(self, p):
        "Value : INTKW '(' String ')'"
        p[0] = f"{p[3]}atoi\n"

    def p_Value_float(self, p):
        "Value : INTKW '(' ExpressionF ')'"
        p[0] = f"{p[3]}ftoi\n"

    def p_ValueF_FLOAT(self, p):
```

```
    "ValueF : FLOAT"
    p[0] = f"pushf {p[1]}\n"

def p_ValueF_IDF(self, p):
    "ValueF : VARF"
    p[0] = f"pushg {self.fp[p[1]][0]}\n"

def p_ValueF_str(self, p):
    "ValueF : FLOATKW '(' String ')'"
    p[0] = f"{p[3]}atof\n"

def p_ValueF_int(self, p):
    "ValueF : FLOATKW '(' Expression ')'"
    p[0] = f"{p[3]}itof\n"

def p_Value_ArrayElem(self, p):
    "Value : VARA '[' Expression ']'"
    p[0] = f"pushgp\npushi {self.fp[p[1]][0]}\npadd\n{p[3]}loadn\n"

def p_Value_Array2dElem(self, p):
    "Value : VARA '[' Expression ']' '[' Expression ']'"
    p[0] = f"pushgp\npushi {self.fp[p[1]][0]}\npadd\n{p[3]}pushi
    ↪  {self.fp[p[1]][2][1]}\nmul\n{p[6]}add\nloadn\n"

def p_Logical_Comparisons(self, p):
    """Logical : Expression '>' Expression
               | Expression '<' Expression
               | Expression GE Expression
               | Expression LE Expression
               | Expression EQ Expression
               | Expression NE Expression"""
    if p[2] == '>':
        p[0] = p[1] + p[3] + "sup\n"
    elif p[2] == '<':
        p[0] = p[1] + p[3] + "inf\n"
    elif p[2] == '>=':
        p[0] = p[1] + p[3] + "supeq\n"
    elif p[2] == '<=':
        p[0] = p[1] + p[3] + "infeq\n"
    elif p[2] == '==':
        p[0] = p[1] + p[3] + "equal\n"
    elif p[2] == '!=':
        p[0] = p[1] + p[3] + "equal\nnot\n"

def p_Logical_Comparisons_f(self, p):
    """Logical : ExpressionF '>' ExpressionF
               | ExpressionF '<' ExpressionF
               | ExpressionF GE ExpressionF
```

```
                | ExpressionF LE ExpressionF
                | ExpressionF EQ ExpressionF
                | ExpressionF NE ExpressionF"""
    if p[2] == '>':
        p[0] = p[1] + p[3] + "fsup\nftoi\n"
    elif p[2] == '<':
        p[0] = p[1] + p[3] + "finf\nftoi\n"
    elif p[2] == '>=':
        p[0] = p[1] + p[3] + "fsupeq\nftoi\n"
    elif p[2] == '<=':
        p[0] = p[1] + p[3] + "finfeq\nftoi\n"
    elif p[2] == '==':
        p[0] = p[1] + p[3] + "equal\n"
    elif p[2] == '!=':
        p[0] = p[1] + p[3] + "equal\nnot\n"

def p_Logical_Comparisons_f2(self, p):
    """Logical : ExpressionF '>' Expression
                | ExpressionF '<' Expression
                | ExpressionF GE Expression
                | ExpressionF LE Expression
                | ExpressionF EQ Expression
                | ExpressionF NE Expression"""
    p[0] = p[1] + p[3] + "itof\n"
    if p[2] == '>':
        p[0] += "fsup\nftoi\n"
    elif p[2] == '<':
        p[0] += "finf\nftoi\n"
    elif p[2] == '>=':
        p[0] += "fsupeq\nftoi\n"
    elif p[2] == '<=':
        p[0] += "finfeq\nftoi\n"
    elif p[2] == '==':
        p[0] += p[1] + p[3] + "equal\n"
    elif p[2] == '!=':
        p[0] += "equal\nnot\n"

def p_Logical_Comparisons_f3(self, p):
    """Logical : Expression '>' ExpressionF
                | Expression '<' ExpressionF
                | Expression GE ExpressionF
                | Expression LE ExpressionF
                | Expression EQ ExpressionF
                | Expression NE ExpressionF"""
    p[0] = p[1] + "itof\n" + p[3]
    if p[2] == '>':
        p[0] += "fsup\nftoi\n"
    elif p[2] == '<':
```

```
            p[0] += "finf\nftoi\n"
        elif p[2] == '>=':
            p[0] += "fsupeq\nftoi\n"
        elif p[2] == '<=':
            p[0] += "finfeq\nftoi\n"
        elif p[2] == '==':
            p[0] += p[1] + p[3] + "equal\n"
        elif p[2] == '!=':
            p[0] += "equal\nnot\n"


    def p_Logical_Parens(self, p):
        "Logical : '(' Logical ')'"
        p[0] = p[2]

    def p_Logical_AND(self, p):
        "Logical : Logical AND Logical"
        p[0] = p[1] + p[3] + "mul\n"

    def p_Logical_OR(self, p):
        "Logical : Logical OR Logical"
        p[0] = p[1] + p[3] + "add\n"

    def p_Logical_NOT(self, p):
        "Logical : NOT Logical"
        p[0] = p[2] + "not\n"

    def p_Boolean(self, p):
        """Boolean : Expression
                   | Logical"""
        p[0] = p[1]

    def p_String(self, p):
        "String : TEXT"
        p[0] = "pushs " + '"' + p[1].strip('"') + '"' + "\n"

    def p_String_var(self, p):
        "String : VARS"
        p[0] = f"pushg {self.fp[p[1]][0]}\n"

    def p_String_input(self, p):
        "String : INPUT '(' String ')'"
        p[0] = f"{p[3]}writes\nread\n"

    def p_String_input_empty(self, p):
        "String : INPUT '(' ')'"
        p[0] = f"read\n"
```

```
    def p_Empty(self, p):
        "Empty : "
        pass

    def p_error(self, p):
        print(f"Syntax error - unexpected token '{p.value}' on line {p.lineno}.")


    def build(self, **kwargs):
        self.fp = dict()
        self.stack_size = 0
        self.lexer = Lexer(self.fp)
        self.lexer.build()
        self.parser = yacc.yacc(module=self, **kwargs)
        self.labels_if = 0
        self.labels_if_else = 0
        self.labels_cycle = 0
        self.labels_cycle_exit = 0
        self.labels_pow = 0
```