Processamento de Linguagens (3º ano de MIEINF)

Trabalho Prático 2

Relatório de Desenvolvimento

Luis Pereira	Carlos Afonso	Gonçalo Nogueira
(a77667)	(a82529)	(a86617)

8 de junho de 2021

Resumo

O presente relatório descreve o segundo trabalho prático realizado no âmbito da disciplina de *Processamento de Linguagens*, ao longo do segundo semestre do terceiro ano do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

O objetivo deste segundo trabalho prático é o desenvolvimento de um compilador para uma linguagem por nós definida capaz de reconhecer a sintaxe e gerar um código máquina, um *"pseudo"Assembly*, que através do uso de uma máquina de stack virtual fornecida pelos docentes efetua as operações pretendidas.

Este relatório inicia-se com uma breve contextualização, seguindo-se de uma descrição dos vários pontos a desenvolver e explicação da estratégia utilizada para responder a cada um dos problemas colocados. Por fim, são apresentadas conclusões sobre o trabalho realizado.

Conteúdo

Capítulo 1

Introdução

1.1 Contextualização

O presente relatório foi elaborado para o segundo trabalho prático da unidade curricular de Processamento de Linguagens, inserida no 2º semestre do 3º ano do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

1.2 Objetivos e trabalho proposto

Pretende-se com este relatório detalhar todo o processo da criação de um compilador desenvolvido com auxílio dos módulos **LEX/YACC** do **PLY/Python** baseado numa gramática independente do contexto capaz de lidar com todas funcionalidades propostas no enunciado.

1.3 Resumo do trabalho a desenvolver

Para atingir os objetivos propostos do trabalho, foi importante perceber primeiro como funciona a máquina de stack virtual, os comandos utilizados bem como as ações por eles despoletadas.

Capítulo 2

Análise e Especificação

2.1 Plano de implementação

Após a análise do problema do enunciado e das respectivas tarefas que o mesmo propunha, o grupo decidiu por procurar construir uma gramática robusta e suficientemente intuitiva para o utilizador, mas capaz de reconhecer todo o tipo de elementos necessários para responder aos problemas colocados. Seguidamente, um analisador sintático terá de ser criado para reconhecer a linguagem e um parser para transformar o que foi escrito na nossa linguagem para código máquina.

2.2 Descrição do Problema

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 controlo do fluxo de execução.
- Efetuar instruções cíclicas para controlo do fluxo de execução, permitindo o seu aninhamento (for-do).

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.

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 impares de uma sequência de números naturais.
- Ler e armazenar N números num array; imprimir os valores por ordem inversa.
- Invocar e usar num programa seu uma função 'potencia()', que começa por ler do input a base B e o expoente E e retorna o valor B^E.

Capítulo 3

Concepção/desenho da Resolução

3.1 Analisador Léxico

Para começar a desenvolver a nossa gramática, inciou-se o processo de criação de literais a ser usados e também os tokens necessários precedidos das suas respetivas espressões regulares que permite ao analisador efetuar o seu reconhecimento.

3.1.1 Expressões Regulares

Foram criadas então as seguintes expressões regulares que reconhecem os tokens.

NUM

O token representa um número inteiro e tem a seguinte expressão regular. **r'\d+'**

FNAM

O token representa o nome de uma função que tem de ser iniciada por uma letra maiúscula e tem a seguinte expressão regular. **r'[A-Z]\w*'**

INT

O token representa o início de uma declaração de uma variável do tipo inteiro e tem a seguinte expressão regular. **r'Int'**

VAR

O token representa o nome de uma variável e tem a seguinte expressão regular. **r'[a-z]+\d*'**

RUN

O token representa a chamada do comando run e tem a seguinte expressão regular. **r'run'**

READ

O token representa a chamada do comando read e tem a seguinte expressão regular. **r'read'**

WRITE

O token representa a chamada do comando write e tem a seguinte expressão regular. **r'write'**

IF

O token representa o início de um ciclo if e tem a seguinte expressão regular. **r'if'**

ELSE

O token representa o else caso a condição do if não se verifique e tem a seguinte expressão regular. **r'else'**

FOR

O token representa o inicio do ciclo for e tem a seguinte expressão regular. **r'for'**

BOOL

O token representa um bool que pode ser *true* ou *false* e tem a seguinte expressão regular. **r'(False|True)'**

BOOLEAN

O token representa o início de uma declaração de variável do tipo booleano e tem a seguinte expressão regular. **r'Boolean'**

OR

O token representa a disjunção e tem a seguinte expressão regular. **r'or'**

AND

O token representa a conjunção e tem a seguinte expressão regular. **r'and'**

MOD

O token representa a operação módulo tem a seguinte expressão regular. **r'mod'**

EQUALS

O token representa a igualdade e tem a seguinte expressão regular. **r'=='**

INF

O token representa a inferioridade tem a seguinte expressão regular. **r'¡'**

SUP

O token representa a superioridade tem a seguinte expressão regular. **r'¿'**

INFEQ

O token representa a inferioridade ou igualdade e tem a seguinte expressão regular. **r'¡='**

SUPEQ

O token representa a superioridade ou igualdade e tem a seguinte expressão regular. **r'¿='**

RETURN

O token representa a chamada do comando return e tem a seguinte expressão regular. **r'return'**

TEXTO

O token representa uma *string* e tem a seguinte expressão regular. **r'"[A-Za-z]*"'**

3.2 Compilador

3.2.1 Gramática Desenvolvida

```
Rule 0     S' -> Call
Rule 1     Call -> Elementos RUN ( ) Elementos
Rule 2     Elementos -> Elemento Elementos
Rule 3     Elementos -> <empty>
Rule 4     Elemento -> OperAr
Rule 5     Elemento -> OperLog
Rule 6     Elemento -> OperRel
Rule 7     Elemento -> Atri
Rule 8     Elemento -> RETURN Var
Rule 9     Elemento -> Funcao
Rule 10    Elemento -> Io
Rule 11    Elemento -> Conditional
Rule 12    Elemento -> For fimFor fimFor2
Rule 13    For -> FOR ( Atri ; OperRel
Rule 14    fimFor -> ; Atri )
Rule 15    fimFor2 -> { Elementos }
Rule 16    Conditional -> inicioElse fimElse
Rule 17    inicioElse -> ELSE
Rule 18    fimElse -> { Elementos }
Rule 19    Conditional -> inicioIF fimIF
Rule 20    fimIF -> { Elementos }
Rule 21    inicioIF -> IF ( OperRel )
Rule 22    inicioIF -> IF ( Var )
Rule 23    Funcao -> FNAM ( ) { Elementos }
Rule 24    Atri -> VAR [ NUM ]
Rule 25    Atri -> VAR [ NUM ] = NUM
Rule 26    Atri -> VAR [ NUM ] = VAR
Rule 27    Atri -> VAR [ Var ] = NUM
Rule 28    Atri -> VAR [ Var ] = VAR
Rule 29    Atri -> INT Var = VAR [ Var ]
Rule 30    Atri -> INT Var = Funcao
Rule 31    Atri -> INT Var = OperAr
Rule 32    Atri -> INT Var = NUM
Rule 33    Atri -> BOOLEAN Var = OperLog
Rule 34    Atri -> BOOLEAN Var = OperRel
Rule 35    Atri -> BOOLEAN Var = BOOL
Rule 36    Io -> READ ( Var )
Rule 37    Io -> WRITE NUM
Rule 38    Io -> WRITE Var
Rule 39    Io -> WRITE ( TEXTO )
Rule 40    OperLog -> BOOL AND BOOL
Rule 41    OperLog -> Var AND Var
Rule 42    OperLog -> Var AND BOOL
Rule 43    OperLog -> BOOL AND Var
Rule 44    OperLog -> BOOL OR BOOL
Rule 45    OperLog -> Var OR Var
Rule 46    OperLog -> Var OR BOOL
Rule 47    OperLog -> BOOL OR Var
Rule 48    OperAr -> OperAr + Termo
Rule 49    OperAr -> OperAr - Termo
```

```
Rule 50     OperAr -> Termo
Rule 51     Termo -> Termo * Factor
Rule 52     Termo -> Termo / Factor
Rule 53     Termo -> Termo MOD Factor
Rule 54     Termo -> Factor
Rule 55     Factor -> ( OperAr )
Rule 56     Factor -> NUM
Rule 57     Factor -> Var
Rule 58     OperRel -> Factor EQUALS Factor
Rule 59     OperRel -> Factor INF Factor
Rule 60     OperRel -> Factor SUP Factor
Rule 61     OperRel -> Factor INFEQ Factor
Rule 62     OperRel -> Factor SUPEQ Factor
Rule 63     Var -> VAR
```

Decisões da gramática

3.2.2 Regras de tradução

Inicialmente irá explicar-se os métodos auxiliares criados bem como os registos criados para guardar informações. Relativamente a métodos auxiliares foi criado apenas o *line_prepender* que escreve no início de um ficheiro obrigando a todo o conteúdo que já lá se encontrava a mover-se para baixo.

```
def line_prepender(line):
    with open("demofile.vm", 'r+') as f:
        content = f.read()
        f.seek(0, 0)
        f.write(line.rstrip('\r\n') + '\n' + content)
```

Os registos criados foram os seguintes:

```
parser.registers = {}
parser.registers.update({"offset":0})
parser.registers.update({"contador":0})
parser.registers.update({"contador2":0})
parser.registers.update({"contador3":0})
parser.registers.update({"contador4":0})
parser.registers.update({"contador5":0})
parser.registers.update({"for":0})
parser.registers.update({"for2":0})
parser.registers.update({"for3":0})
parser.registers.update({"varCount":0})
parser.registers.update({"arrayCount":0})
```

Dentro do dicionário foram criados vários contadores, todos iniciados a 0, existindo 5 contadores a serem utilizados nos condicionais, nos ciclos for, um contador designado de *offset* para lidar com os identificadores das variáveis e 2 contadores para lidar com o número de variaveis e de *arrays*. De salientar que o *offset* é utilizado para criar pares chaves valor onde a chave é o nome de uma variável declarada no programa e o valor é o seu identificador. Sempre que uma nova variável é declarada o contador é incrementado.

```
def p_Call(p):
    "Call : Elementos RUN '(' ')' "
    f.write(p[1]+p[5])
    while p.parser.registers.get("varCount") > 0:
        line_prepender("PUSHI 0\n")
        p.parser.registers.update({"varCount" : p.parser.registers.get("varCount") -1})
    if( p.parser.registers.get("arrayCount") >0):
        line_prepender("PUSHN "+ str(p.parser.registers.get("arrayCount")))
```

Esta é a nossa produção 1, ou seja, o axioma da nossa linguagem. Optamos por obrigar á declaração do comando **run()** para correr um programa.

É nesta produção onde se efetua a escrita do código máquina para um ficheiro, indo buscar tudo o que vem dos Elementos que estão antes da declaração do run. É também nesta produção onde se escreve no início do ficheiro **PUSHI 0** quantas vezes o número de variáveis declaradas no programa e também o **PUSHN** reservando espaço na stack consoante o tamanho dos arrays declarados no programa.

```
def p_Elementos(p):
    "Elementos : Elementos Elemento"
    p[0] = p[1] + p[2]

def p_Elementos_vazio(p):
    "Elementos : "
    p[0]= ""
```

O símbolo Elementos pode ser constituído por um ou mais Elemento agrupando no p[0] todos os valores de cada elemento individual ou pode ser vazio.

```
def p_Elemento_OperAr(p):
    "Elemento : OperAr"
    p[0]=p[1]

def p_Elemento_OperLog(p):
    "Elemento : OperLog"
    p[0]=p[1]

def p_Elemento_OperRel(p):
    "Elemento : OperRel"
    p[0]=p[1]
```

O símbolo Elemento representa uma qualquer componente implementada na nossa linguagem podendo ser operações aritméticas, lógicas ou relacionais.

```
def p_Elemento_Atri(p):
    "Elemento : Atri"
    p[0]=p[1]
```

Além de operações um Elemento pode ser uma atribuição.

```
def p_Elemento_Return(p):
    "Elemento : RETURN Var"
    p[0]= ("PUSHG " + str(p.parser.registers.get(p[2])) + "\n")
```

Pode ser um retornar de uma variável e sendo assim procura-se o identificador da variável aos registos e fazer **PUSHG** desse identificador, colocando assim o valor guardado no início da pilha.

```
def p_Elemento_Funcao(p):
    "Elemento : Funcao "
    p[0]=p[1]
```

Pode ser uma declaração de uma função.

```
def p_Elemento_Io(p):
    "Elemento : Io "
    p[0]=p[1]
```

Pode ser uma operação IO.

```
def p_Elemento_Conditional(p):
    "Elemento : Conditional "
    p[0]=p[1]
```

Pode ser um condicional.

```
def p_Elemento_For(p):
    "Elemento : For  fimFor fimFor2 "
    p[0]=p[1] + p[3] + p[2]
```

Pode ser um ciclo for onde neste caso inicialmente é compilado o símbolo não terminal for, sendo precedido do fimFor2 e do fimFor.
Nesta produção a ordem é importante devido á estratégia de implementação do ciclo for em código máquina. Esta forma de ordem deve-se ao facto da atribuição final do for ser apenas feita depois de ja ter processado o corpo do ciclo.

```
def p_For(p):
    "For : FOR '(' Atri ';' OperRel  "
    p[0] = p[3] +("FOR"+str(p.parser.registers.get("for"))+":\n")+ p[5]  +
    ("JZ fimFOR"+str(p.parser.registers.get("for2"))+"\n")
    p.parser.registers.update({"for": (p.parser.registers.get("for")+1) })
```

No caso da declaração do ciclo é obrigatório ser lido o token FOR, é reconhecido a atribuição inicial, é declarada a label *FOR* com o valor no contador *for* para marcar o início do ciclo no código máquina. Posteriormente é reconhecida a operação relacional e é efetuado o comando **JZ** com a label *fimFor* juntamente com o contador para estes saltos. Caso o valor da operação relacional seja 0 vai-se efetuar um salto para o local marcado pela label. O contador das labels para o início de um ciclo for é incrementado.

```
def p_fimFor2(p):
    "fimFor2 : '{' Elementos  '}'"
    p[0] = p[2]
```

Este é a produção onde são reconhecidos todos os elementos constituíntes do corpo do ciclo for. Esta parte é efetuada depois do resultado da tradução da produção anterior.

```
def p_fimFor(p):
    "fimFor : ';' Atri ')'  "
    p[0] = p[2] + ("JUMP FOR"+str(p.parser.registers.get("for3"))+"\n") +
    ("fimFOR"+str(p.parser.registers.get("for2"))+":\n")
    p.parser.registers.update({"for2": (p.parser.registers.get("for2")+1) })
    p.parser.registers.update({"for3": (p.parser.registers.get("for3")+1) })
```

Esta produção lida com a atribuição da última parte do for. Esta ultima parte efetua o reconhecimento e tradução da atribuição, introduz-se o comando **JUMP** para a label do respetivo for usando para isso o contador *for3*, obrigando ao salto para o início do ciclo for. Por fim, indica-se a label *fimFOR* para indicar no código máquina a terminação de um ciclo for.

```
def p_Conditional_IF(p):
    "Conditional : inicioIF fimIF"
    p[0] = p[1]+p[2]

def p_Conditional_Else(p):
    "Conditional : inicioElse fimElse"
    p[0] = p[1]+p[2]
```

Estas são as produções que lidam com os condicionais, podendo ser referentes ao *if* ou referentes ao *else*.

```
def p_Inicio_If_Oper(p):
    "inicioIF : IF '(' OperRel ')' "
    p[0] = p[3] + ("JZ ELSE"+str(p.parser.registers.get("contador"))+"\n")
    p.parser.registers.update({"contador": (p.parser.registers.get("contador")+1) })
```

Esta produção indica o início do if, onde neste caso é efetuada uma operação relacional. Essa operação é traduzida e é introduzido o comando **JZ** com a label *ELSE* com o contador. Este salto é verificado quando o resultado é 1 na operação relacional. O contador é incrementado.

```
def p_Inicio_If_Log(p):
    "inicioIF : IF '(' Var ')' "
    p[0] = ("PUSHG "+str(p.parser.registers.get(p[3]))+"\n")+
    ("JZ ELSE"+str(p.parser.registers.get("contador"))+"\n")
    p.parser.registers.update({"contador": (p.parser.registers.get("contador")+1) })
```

Pode existir também o caso onde é comparado o valor de uma variável, neste caso usa-se o comando **PUSHG** com o identificador da variável nos registos e é efetuado o mesmo procedimento que a produção anterior.

```
def p_Conditional_fimIf(p):
    "fimIF :  '{' Elementos '}' "
    p[0] = p[2]+("JUMP fimIF"+str(p.parser.registers.get("contador4"))+"\n")
    p.parser.registers.update({"contador4": (p.parser.registers.get("contador4")+1) })

def p_Conditional_inicioElse(p):
    "inicioElse : ELSE "
    p[0]= ("ELSE"+str(p.parser.registers.get("contador2"))+":\n")
```

Relativamente ao início do else, só acontece se for reconhecido o token *ELSE* e gera uma label **ELSE** com o valor no contador2.

```
def p_Conditional_fimElse(p):
    "fimElse : '{' Elementos '}' "
    p.parser.registers.update({"contador2": (p.parser.registers.get("contador2")+1) })
    p[0]=p[2]+("fimIF"+str(p.parser.registers.get("contador5"))+":\n")
    p.parser.registers.update({"contador5": (p.parser.registers.get("contador5")+1) })
```

Sobre o fimElse são reconhecidos todos os elememtos do seu corpo e traduzidos. O contador2 é incrementado, para o caso de existir outro else dentro de si mesmo indicar uma label atualizada. É declarada a label *fimIF* com o valor no contador5 que marca o fim do condicional. O contador5 é também incrementado.

```
def p_Funcao(p):
    "Funcao : FNAM '(' ')' '{' Elementos '}'"
    p[0]= p[5]
```

Esta produção representa a declaração de uma função na nossa linguagem, esta função não requer nenhum tipo de argumentos e no seu corpo pode existir todo o tipo de elementos.

```
def p_Array(p):
    "Atri : VAR '[' NUM ']'"
    if p[1] not in p.parser.registers:
        p.parser.registers.update({"arrayCount" : p.parser.registers.get("arrayCount") +
        int(p[3])})
        p.parser.registers.update({p[1]: (p.parser.registers.get("offset")) })
        p.parser.registers.update({"offset":(p.parser.registers.get("offset")+int(p[3]))})
    p[0] = ""
```

Esta produção lida com a declaração de um array com um tamanho indicado em p[3], esta declaração guarda o tamanho do array no contador *arrayCount* para se declarar o comando **PUSHN** desse tamanho no início do ficheiro.

```
def p_Atri_Array_NUM(p):
    "Atri : VAR '[' NUM ']' '=' NUM"
    p[0] =  "PUSHGP\n" + "PUSHI " + str(p.parser.registers.get(p[1])) +"\n" +
    "PADD\n" + "PUSHI " + p[3] + "\n" + "PUSHI " + p[6] + "\n" + "STOREN\n"
```

```
def p_Atri_Array_Var(p):
    "Atri : VAR '[' NUM ']' '=' VAR"
    p[0] =  "PUSHGP\n" + "PUSHI " + str(p.parser.registers.get(p[1])) +"\n" +
    "PADD\n" + "PUSHI " + p[3] + "\n" + "PUSHG " +
    str(p.parser.registers.get(p[6])) + "\n" + "STOREN\n"

def p_Atri_Array_Var_NUM(p):
    "Atri : VAR '[' Var ']' '=' NUM"
    p[0] =  "PUSHGP\n" + "PUSHI " + str(p.parser.registers.get(p[1])) +"\n" +
    "PADD\n" + "PUSHG " + str(p.parser.registers.get(p[3])) + "\n" +
    "PUSHI " + p[6] + "\n" + "STOREN\n"

def p_Atri_Array_Var_Var(p):
    "Atri : VAR '[' Var ']' '=' VAR"
    p[0] =  "PUSHGP\n" + "PUSHI " + str(p.parser.registers.get(p[1])) +"\n" +
    "PADD\n" + "PUSHG " + str(p.parser.registers.get(p[3])) + "\n" +
    "PUSHG " + str(p.parser.registers.get(p[6])) + "\n" + "STOREN\n"
```

Estas últimas 4 produções representam todas a introdução ou alteração de um valor num determinado índice do array. Foram feitas estas produções para lidar com a possibilidade de tanto o índice como o valor a inserir serem variáveis ou números. Este facto afeta o tipo de tratamento nas produções pois no caso das variáveis é necessário fazer **PUSHG** do seu identificador para colocar o valor da variável no início da pilha e no caso de ser um número é apenas **PUSHI** desse número.
Relativamente á estratégia para lidar com os arrays, é efetuado primeiramente o comando **PUSHGP** que empilha o valor do registo *gp*, depois coloca-se o valor do identificador da variável que nomeia o array na pilha e usa-se o comando **PADD** que retira da stack o inteiro mais o endereço de gp e empilha o endereço somado com o inteiro. Depois disso empilha-se o valor do índice e o valor da atribuição e com o comando **STOREN** guarda-se o valor no índice pretendido.

```
def p_Atri_Var_Array(p):
    "Atri : INT Var '=' VAR '[' Var ']'"
    if p[2] in p.parser.registers:
        p[0] =  "PUSHGP\n" + "PUSHI " + str(p.parser.registers.get(p[4])) +"\n" +
        "PADD\n" + "PUSHG " + str(p.parser.registers.get(p[6])) + "\n" +
        "LOADN\n"  + ("STOREG " +
        str(p.parser.registers.get(p[2]))+"\n")
    else:
        p.parser.registers.update({p[2]:p.parser.registers.get("offset")})
        p[0] =  "PUSHGP\n" + "PUSHI " + str(p.parser.registers.get(p[4])) +"\n" +
        "PADD\n" + "PUSHG " + str(p.parser.registers.get(p[6])) + "\n" +
        "LOADN\n"  + ("STOREG " + str(p.parser.registers.get(p[2]))+"\n")
        p.parser.registers.update({"offset": (p.parser.registers.get("offset")+1) })
```

Esta produção foi criada para lidar com a atribuição de um valor guardado num array a uma variável. A estratégia é bastante semelhante ás produções mencionadas anteriormente diferindo apenas no final, onde se utiliza o comando **LOADN** para empilhar o valor guardado na stack e por fim um **STOREG** com o identificador da variável para guardar o número nesse registo.

```
def p_Atri_Funcao(p):
    "Atri : INT Var '=' Funcao "
    if p[2] in p.parser.registers:
        p[0] = p[4]+("STOREG " + str(p.parser.registers.get(p[2]))+"\n")
    else:
        p.parser.registers.update({p[2]:p.parser.registers.get("offset")})
        p[0] = p[4]+("STOREG " + str(p.parser.registers.get(p[2]))+"\n")
        p.parser.registers.update({"offset": (p.parser.registers.get("offset")+1) })
```

Esta produção atribui o resultado da execução de uma função e guarda o valor na variável declarada. Para isto funcionar é necessário existir um *return* de um inteiro no fim do corpo da função para ser possível guardá-lo.

```
def p_Atri_Exp(p):
    "Atri : INT Var '=' OperAr"
    if p[2] in p.parser.registers:
        p[0] = p[4]+("STOREG " + str(p.parser.registers.get(p[2]))+"\n")
    else:
        p.parser.registers.update({p[2]:p.parser.registers.get("offset")})
        p[0] = p[4]+("STOREG " + str(p.parser.registers.get(p[2]))+"\n")
        p.parser.registers.update({"offset": (p.parser.registers.get("offset")+1) })

def p_Atri_Simple(p):
    "Atri : INT Var '=' NUM"
    if p[2] in p.parser.registers:
        p[0] = ("PUSHI " +p[4]+"\n") + ("STOREG " +
        str(p.parser.registers.get(p[2]))+"\n")
    else:
        p.parser.registers.update({p[2]:p.parser.registers.get("offset")})
        p[0] = ("PUSHI " +p[4]+"\n") + ("STOREG " +
        str(p.parser.registers.get(p[2]))+"\n")
        p.parser.registers.update({"offset": (p.parser.registers.get("offset")+1) })
```

Estas 2 produções são bastante semelhantes na medida em que se atribui a uma variável ou o resultado de uma operação aritmética ou um número diretamente. No primeiro caso, o resultado da operação já se encontra na pilha bastando apenas usar o **STOREG** com o identificador da variável para atribuir o número á variável. No segundo caso, basta apenas efetuar um **PUSHI** do número sendo o próximo passo igual á primeira produção. De salientar, e como é habitual ao longo do processo de tradução que caso a variável ainda não tenha sido declarada previamente, é criado um novo par chave-valor que contém o nome da variável bem como o seu identificador que se incrementa sempre que um novo par é introduzido.

```
def p_Atri_Log_Oper(p):
    "Atri : BOOLEAN Var '=' OperLog"
    if p[2] in p.parser.registers:
        p[0]= p[4]+("STOREG " + str(p.parser.registers.get(p[2]))+"\n")
    else:
```

```
        p.parser.registers.update({p[2]:p.parser.registers.get("offset")})
        p[0]=p[4]+("STOREG " + str(p.parser.registers.get(p[2]))+"\n")
        p.parser.registers.update({"offset": (p.parser.registers.get("offset")+1) })

def p_Atri_Rel_Oper(p):
    "Atri : BOOLEAN Var '=' OperRel"
    if p[2] in p.parser.registers:
        p[0]= p[4]+("STOREG " + str(p.parser.registers.get(p[2]))+"\n")
    else:
        p.parser.registers.update({p[2]:p.parser.registers.get("offset")})
        p[0]=p[4]+("STOREG " + str(p.parser.registers.get(p[2]))+"\n")
        p.parser.registers.update({"offset": (p.parser.registers.get("offset")+1) })

def p_Atri_Log(p):
    "Atri : BOOLEAN Var '=' BOOL"
    if(p[4]=="True"):
        bol = "1"
    else:
        bol = "0"
    if p[2] in p.parser.registers:
        p[0] = ("PUSHI "+bol+"\n")+("STOREG " + str(p.parser.registers.get(p[2]))+"\n")
    else:
        p.parser.registers.update({p[2]:p.parser.registers.get("offset")})
        p[0] = ("PUSHI "+bol+"\n")+("STOREG " + str(p.parser.registers.get(p[2]))+"\n")
        p.parser.registers.update({"offset": (p.parser.registers.get("offset")+1) })
```

Estas 3 produções são quase as mesmas que as 2 anteriores diferindo apenas no tipo de variável que é declarada, no entanto, para o código maquina, Verdadeiro e Falso não existe, sendo apenas 0 e 1, o que torna a tradução semelhante á tradução das variáveis do tipo inteiro e que já foram mencionadas e detalhadas anteriormente.

```
def p_Io_Read_Int(p):
    "Io : READ '(' Var ')'"
    if p[3] in p.parser.registers:
        p[0] = ('READ\nATOI\n')  + ("STOREG ") +
        (str(p.parser.registers.get(p[3])) + "\n")

    else:
        p.parser.registers.update({p[3]:p.parser.registers.get("offset")})
        p[0] = ('READ\nATOI\n')  + ("STOREG ") +
        (str(p.parser.registers.get(p[3])) + "\n")
        p.parser.registers.update({"offset": (p.parser.registers.get("offset")+1) })
```

Nesta produção lida-se com a leitura e posterior atribuição do valor introduzido á variável indicada. Para se introduzir um valor é efetuado o comando **READ** que pede um input ao utilizador precedido do **ATOI** que transforma a string num valor inteiro deixando na pilha o valor do tipo inteiro. Por fim, usa-se o **STOREG** com o identificador da variável para guardar o valor nessa variável.

```
def p_Io_Write_Int(p):
    "Io : WRITE NUM"
    p[0] = ("PUSHI "+p[2]+"\n")+('WRITEI\n')

def p_Io_Write_Var(p):
    "Io : WRITE Var"
    p[0] = ("PUSHG " + str(p.parser.registers.get(p[2])) + "\n") + ('WRITEI\n')

def p_Io_Write_String(p):
    "Io : WRITE '(' TEXTO ')' "
    p[0] = ("PUSHS "+p[3]+"\n")+('WRITES\n')
```

Nestas 3 produções lida-se com o *write* da nossa linguagem que serve para colocar algo no output, podendo ser diretamente um número ou string ou o valor de uma variável. No primeiro caso, dá-se **PUSHI** do número e depois usa-se o **WRITEI** para escrever esse número no output. No segundo, faz-se **PUSHG** do identificador da variável para colocar na pilha o número atribuído á variavel e depois o **WRITEI** para escrever o número para o output. Por fim, no terceiro caso usa-se **PUSHS** para colocar a string na pilha e o **WRITES** para escrever a string no output.

```
def p_Oper_AND_Bool(p):
    "OperLog : BOOL AND BOOL"
    if(p[1]=="True"):
        bol = "1"
    else:
        bol = "0"
    if(p[3]=="True"):
        bol2 = "1"
    else:
        bol2 = "0"
    p[0] = ("PUSHI "+bol+"\n")+("PUSHI "+bol2+"\n")+(operacoes.get("*")+"\n")

def p_Oper_AND_Var(p):
    "OperLog : Var AND Var"
    p[0] = ("PUSHG " + str(p.parser.registers.get(p[1])) + "\n")+
    ("PUSHG " + str(p.parser.registers.get(p[3])) + "\n")+(operacoes.get("*")+"\n")

def p_Oper_AND_Var_BOOLEAN(p):
    "OperLog : Var AND BOOL"
    if(p[3]=="True"):
        bol = "1"
    else:
        bol = "0"
    p[0]=("PUSHG " + str(p.parser.registers.get(p[1])) + "\n")+("PUSHI "+bol+"\n")+
    (operacoes.get("*")+"\n")

def p_Oper_AND_BOOLEAN_Var(p):
    "OperLog : BOOL AND Var"
```

```
    if(p[1]=="True"):
        bol = "1"
    else:
        bol = "0"
    p[0]= ("PUSHI "+bol+"\n")+("PUSHG " + str(p.parser.registers.get(p[3])) + "\n")+
    (operacoes.get("*")+"\n")
```

Estas 4 produções tratam da operação lógica de booleanos **And**. Inicialmente averiguar-se o valor lógico, 0 ou 1, das variáveis ou booleanos. Matematicamente a operação **And** pode ser descrita como uma multiplicação dos dois valores lógicos.

$$\begin{aligned} 0 * 0 &= 0 \\ 0 * 1 &= 0 \\ 1 * 0 &= 0 \\ 1 * 1 &= 1 \end{aligned} \tag{3.1}$$

Com o problema formulado, implementamos a operação em código máquina que é atribuída ao a p[0] sob forma de string.

```
def p_Oper_OR_BOOLEAN(p):
    "OperLog : BOOL OR BOOL"
    if(p[1]=="True"):
        bol = "1"
    else:
        bol = "0"
    if(p[3]=="True"):
        bol2 = "1"
    else:
        bol2 = "0"
    p[0] = ("PUSHI "+bol+"\n")+
    ("PUSHI "+bol2+"\n")+(operacoes.get("+")+"\n")+("PUSHI "+bol+"\n")+
    ("PUSHI "+bol2+"\n")+(operacoes.get("*")+"\n")+
    (operacoes.get("-")+"\n")

def p_Oper_OR_Var(p):
    "OperLog : Var OR Var"
    p[0] =("PUSHG " + str(p.parser.registers.get(p[1])) + "\n")+("PUSHG " +
    str(p.parser.registers.get(p[3])) + "\n")+(operacoes.get("+")+"\n")+("PUSHG " +
    str(p.parser.registers.get(p[1])) + "\n")+("PUSHG " +
    str(p.parser.registers.get(p[3]))+ "\n")+
    (operacoes.get("*")+"\n")+(operacoes.get("-")+"\n")

def p_Oper_OR_Var_Bool(p):
    "OperLog : Var OR BOOL"
    if(p[3]=="True"):
        bol = "1"
```

```
    else:
        bol = "0"
    p[0] = ("PUSHG " + str(p.parser.registers.get(p[1])) + "\n")+("PUSHI "+bol+"\n")+
    (operacoes.get("+")+"\n")+("PUSHG " + str(p.parser.registers.get(p[1])) + "\n")+
    ("PUSHI "+bol+"\n")+(operacoes.get("*")+"\n")+
    (operacoes.get("-")+"\n")

def p_Oper_OR_Bool_Var(p):
    "OperLog : BOOL OR Var"
    if(p[1]=="True"):
        bol = "1"
    else:
        bol = "0"
    p[0]= ("PUSHI "+bol+"\n")+("PUSHG " + str(p.parser.registers.get(p[3])) + "\n")+
    (operacoes.get("+")+"\n")+("PUSHI "+bol+"\n")+ ("PUSHG " +
    str(p.parser.registers.get(p[3])) + "\n")+(operacoes.get("*")+"\n")+
    (operacoes.get("-")+"\n")
```

Estas 4 produções tratam da operação lógica de booleanos **Or**. Inicialmente averiguar-se o valor lógico, 0 ou 1, das variáveis ou booleanos. Matematicamente a operação **Or** pode ser descrita como a subtração entre a soma e a multiplicação das variáveis ou booleanos.

$$\begin{aligned} (0+0)-(0*0)&=0\\ (0+1)-(0*1)&=1\\ (1+0)-(1*0)&=1\\ (1+1)-(1*1)&=1 \end{aligned} \tag{3.2}$$

Com o problema formulado, implementamos a operação em código máquina que é atribuída ao a p[0] sob forma de string.

```
def p_Oper_add(p):
    "OperAr : OperAr '+' Termo"
    p[0]= p[1] + p[3] + "ADD\n"

def p_Oper_sub(p):
    "OperAr : OperAr '-' Termo"
    p[0]= p[1] + p[3] + "SUB\n"
```

Estas produções lidam com as operações aritméticas, onde o *OperAr* e o *Termo* já se encontram traduzidos, bastando apenas usar o comando **ADD** ou **SUB** dependendo do sinal reconhecido.

```
def p_Oper_termo(p):
    "OperAr : Termo"
    p[0]=p[1]

def p_Oper_mul(p):
    "Termo : Termo '*' Factor"
    p[0]= p[1] + p[3] + "MUL\n"
```

```
def p_Termo_div(p):
    "Termo : Termo '/' Factor"
    p[0]= p[1] + p[3] + "DIV\n"

def p_Termo_mod(p):
    "Termo : Termo MOD Factor"
    p[0]= p[1] + p[3] + "MOD\n"
```

Nestas produções efetua-se em primeiro lugar e tal como anteriormente o reconhecimento do *Termo* e do *Factor* e depois os comandos **MUL**, **DIV** ou **MOD** dependendo do que é reconhecido. Todos estes comandos usam como argumento os últimos dois inteiros da pilha, retirando-os da pilha e empilhando o resultado.

```
def p_Termo_factor(p):
    "Termo : Factor"
    p[0]=p[1]

def p_Factor_group(p):
    "Factor : '(' OperAr ')'"
    p[0]= p[2]

def p_Factor_num(p):
    "Factor : NUM"
    p[0] = "PUSHI "+p[1]+"\n"

def p_Factor_id(p):
    "Factor : Var"
    p[0] = "PUSHG "+str(p.parser.registers.get(p[1])) + "\n"
```

Estas 2 produções introduzem na pilha os inteiros utilizados nas operações, podendo ser variáveis e ser preciso ir buscar os valores associados ou podendo ser números diretamente.

```
def p_Oper_Rel_Equal(p):
    "OperRel : Factor EQUALS Factor"
    p[0] = p[1] + p[3] + "EQUAL\n"

def p_Oper_Rel_Inf(p):
    "OperRel : Factor INF Factor"
    p[0] = p[1] + p[3] + "INF\n"

def p_Oper_Rel_Sup(p):
    "OperRel : Factor SUP Factor"
    p[0] = p[1] + p[3] + "SUP\n"

def p_Oper_Rel_InfEQ(p):
    "OperRel : Factor INFEQ Factor"
    p[0] = p[1] + p[3] + "INFEQ\n"
```

```
def p_Oper_Rel_SupEQ(p):
    "OperRel : Factor SUPEQ Factor"
    p[0] = p[1] + p[3] + "SUPEQ\n"
```

Estas 5 produções lidam com as operações relacionais e tal como nas aritméticas, efetua-se primeiro o reconhecimento e colocação na pilha dos valores Factor e são precedidos dos comandos do código máquina que representam os tokens das comparações reconhecidos. Estes comandos podem ser **EQUAL**, **INF**, **SUP**, **INFEQ** e **SUPEQ**.

```
def p_Var(p):
    "Var : VAR"
    if p[1] not in p.parser.registers:
        p.parser.registers.update({"varCount" : p.parser.registers.get("varCount") +1})
    p[0] = p[1]
```

Esta última produção serve para contar o número de variáveis declaradas num programa e coloca esse total no contador *varCount* para serem feitos os **PUSHI 0** consoante esse total lido com o objetivo de reservar espaço na pilha para se guardarem os valores associados ás variáveis.

```
def p_error(p):
    print('Erro sintático: ', p)
    parser.success = False
```

Capítulo 4

Codificação e Testes

4.1 Testes realizados e Resultados

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

4.1.1 Ler 4 números e dizer se podem ser os lados de um quadrado.

```
Conta(){
    read(a)
    read(b)
    read(c)
    read(d)
    if(a==b){
        if(b==c){
            if(c==d){
                write("os valores podem ser lados de um quadrado")
            }else{ write("os valores nao podem ser lados de um quadrado")}
        }else{ write("os valores nao podem ser lados de um quadrado")}
    }else{ write("os valores nao podem ser lados de um quadrado")}


}

run()
```

```
1  PUSHI 0
2  PUSHI 0
3  PUSHI 0
4  PUSHI 0
5  START
6  READ
7  ATOI
8  STOREG 0
9  READ
10 ATOI
11 STOREG 1
12 READ
13 ATOI
14 STOREG 2
15 READ
16 ATOI
17 STOREG 3
18 PUSHG 0
19 PUSHG 1
20 EQUAL
21 JZ ELSE0
22 PUSHG 1
23 PUSHG 2
24 EQUAL
25 JZ ELSE1
26 PUSHG 2
27 PUSHG 3
28 EQUAL
29 JZ ELSE2
30 PUSHS "os valores podem ser lados de um quadrado"
31 WRITES
32 JUMP fimIF0
33 ELSE0:
34 PUSHS "os valores nao podem ser lados de um quadrado"
35 WRITES
36 fimIF0:
37 JUMP fimIF1
38 ELSE1:
39 PUSHS "os valores nao podem ser lados de um quadrado"
40 WRITES
41 fimIF1:
42 JUMP fimIF2
43 ELSE2:
44 PUSHS "os valores nao podem ser lados de um quadrado"
45 WRITES
46 fimIF2:
47 Fim: stop
```

4.1.2 Ler um inteiro N, depois ler N números e escrever o menor deles.

```
Int lower = 0
Int i = 0
Int n = 0
write("Indique a quantidade de numeros que vai inserir ")
read(n)

Lower(){
    for(Int count = 0 ; count < n ; Int count= count +1 ){
        read(i)
        if(count == 0){
            Int lower = i
        } else {}
        if( i < lower){
            Int lower = i
        } else {}
    }
    write("O valor mais pequeno ")
    write lower
}

run()
```

Ficheiro gerado com o código máquina.

```
1  PUSHI 0
2  PUSHI 0
3  PUSHI 0
4  PUSHI 0
5  START
6  PUSHI 0
7  STOREG 0
8  PUSHI 0
9  STOREG 1
10 PUSHI 0
11 STOREG 2
12 PUSHS "Indique a quantidade de numeros que vai inserir "
13 WRITES
14 READ
15 ATOI
16 STOREG 2
17 PUSHI 0
18 STOREG 3
19 FOR0:
20 PUSHG 3
21 PUSHG 2
22 INF
23 JZ fimFOR0
24 READ
25 ATOI
26 STOREG 1
27 PUSHG 3
28 PUSHI 0
29 EQUAL
30 JZ ELSE0
31 PUSHG 1
32 STOREG 0
33 JUMP fimIF0
34 ELSE0:
35 fimIF0:
36 PUSHG 1
37 PUSHG 0
38 INF
39 JZ ELSE1
40 PUSHG 1
41 STOREG 0
42 JUMP fimIF1
43 ELSE1:
44 fimIF1:
45 PUSHG 3
46 PUSHI 1
47 ADD
48 STOREG 3
49 JUMP FOR0
50 fimFOR0:
51 PUSHS "O valor mais pequeno "
52 WRITES
53 PUSHG 0
```

```
54 WRITEI
55 Fim: stop
```

4.1.3 Ler N (constante do programa) números e calcular e imprimir o seu produtório.

```
Int n = 5
Int r = 1
Int a = 0

Prod(){
    for(Int i = 0 ; i< n ; Int i = i + 1){
        read(a)
        Int r = r*a
    }
        write("O produtorio dos cinco numeros e ")
        write r
}

run()
```

Ficheiro gerado com o código máquina.

```
1  PUSHI 0
2  PUSHI 0
3  PUSHI 0
4  PUSHI 0
5  START
6  PUSHI 5
7  STOREG 0
8  PUSHI 1
9  STOREG 1
10 PUSHI 0
11 STOREG 2
12 PUSHI 0
13 STOREG 3
14 FOR0:
15 PUSHG 3
16 PUSHG 0
17 INF
18 JZ fimFOR0
19 READ
20 ATOI
21 STOREG 2
22 PUSHG 1
23 PUSHG 2
24 MUL
25 STOREG 1
26 PUSHG 3
27 PUSHI 1
28 ADD
29 STOREG 3
30 JUMP FOR0
31 fimFOR0:
32 PUSHS "O produtorio dos cinco numeros e "
33 WRITES
34 PUSHG 1
35 WRITEI
36 Fim: stop
```

4.1.4 Contar e imprimir os números impares de uma sequência de números naturais.

```
Int counter = 0
Int a = 0
Int n = 0
read(n)


IsOdd(){
    for(Int i = 0; i < n; Int i = i + 1){
        read(a)
        Int r = a mod 2
        if( r == 1){
            write a
            Int counter = counter + 1
        } else {}

    }
    write counter
}

run()
```

Ficheiro gerado com o código máquina.

```
1  PUSHI 0
2  PUSHI 0
3  PUSHI 0
4  PUSHI 0
5  PUSHI 0
6  START
7  PUSHI 0
8  STOREG 0
9  PUSHI 0
10 STOREG 1
11 PUSHI 0
12 STOREG 2
13 READ
14 ATOI
15 STOREG 2
16 PUSHI 0
17 STOREG 3
18 FOR0:
19 PUSHG 3
20 PUSHG 2
21 INF
22 JZ fimFOR0
23 READ
24 ATOI
25 STOREG 1
26 PUSHG 1
27 PUSHI 2
28 MOD
29 STOREG 4
30 PUSHG 4
31 PUSHI 1
32 EQUAL
33 JZ ELSE0
34 PUSHG 1
35 WRITEI
36 PUSHG 0
37 PUSHI 1
38 ADD
39 STOREG 0
40 JUMP fimIF0
41 ELSE0:
42 fimIF0:
43 PUSHG 3
44 PUSHI 1
45 ADD
46 STOREG 3
47 JUMP FOR0
48 fimFOR0:
49 PUSHG 0
50 WRITEI
51 Fim: stop
```

4.1.5 Ler e armazenar N números num array; imprimir os valores por ordem inversa.

```
array[5]

Inverso(){
   for(Int i = 0; 5>i; Int i = i + 1 ){
      read(a)
      array[i] = a
   }

   for(Int i = 4; i>=0; Int i = i - 1){
      Int a = array[i]
      write a
   }

}

run()
```

Ficheiro gerado com o código máquina.

```
1  PUSHN 5
2  PUSHI 0
3  PUSHI 0
4  START
5  PUSHI 0
6  STOREG 5
7  FOR0:
8  PUSHI 5
9  PUSHG 5
10 SUP
11 JZ fimFOR0
12 READ
13 ATOI
14 STOREG 6
15 PUSHGP
16 PUSHI 0
17 PADD
18 PUSHG 5
19 PUSHG 6
20 STOREN
21 PUSHG 5
22 PUSHI 1
23 ADD
24 STOREG 5
25 JUMP FOR0
26 fimFOR0:
27 PUSHI 4
28 STOREG 5
29 FOR1:
30 PUSHG 5
31 PUSHI 0
32 SUPEQ
33 JZ fimFOR1
34 PUSHGP
35 PUSHI 0
36 PADD
37 PUSHG 5
38 LOADN
39 STOREG 6
40 PUSHG 6
41 WRITEI
42 PUSHG 5
43 PUSHI 1
44 SUB
45 STOREG 5
46 JUMP FOR1
47 fimFOR1:
48 Fim: stop
```

4.1.6 Invocar e usar num programa seu uma função 'potencia()', que começa por ler do input a base B e o expoente E e retorna o valor B^E .

```
Int potencia = Potencia(){
   read(b)
   read(c)
   Int d = 1

   for(Int c = c;c>0;Int c = c - 1){
      Int d = d * b
   }
   return d
}

Resultado(){
   Int h = potencia + 1
   write h
}

run()
```

Ficheiro gerado com o código máquina.

```
1  PUSHI 0
2  PUSHI 0
3  PUSHI 0
4  PUSHI 0
5  PUSHI 0
6  START
7  READ
8  ATOI
9  STOREG 0
10 READ
11 ATOI
12 STOREG 1
13 PUSHI 1
14 STOREG 2
15 PUSHG 1
16 STOREG 1
17 FOR0:
18 PUSHG 1
19 PUSHI 0
20 SUP
21 JZ fimFOR0
22 PUSHG 2
23 PUSHG 0
24 MUL
25 STOREG 2
26 PUSHG 1
27 PUSHI 1
28 SUB
29 STOREG 1
30 JUMP FOR0
31 fimFOR0:
32 PUSHG 2
33 STOREG 3
34 PUSHG 3
35 PUSHI 1
36 ADD
37 STOREG 4
38 PUSHG 4
39 WRITEI
40 Fim: stop
```

Capítulo 5

Conclusão

Após a conclusão do trabalho proposto, o grupo avalia o resultado final como satisfatório, uma vez que todos os objetivos foram atingidos. Ao longo do desenvolvimento do compilador o grupo reforçou os conhecimentos sobre o *LEX/YACC* do *PLY/PYTHON* adquiridos na disciplina e voltou a relembrar como funciona código máquina ainda que tenha sido um pseudo código foi algo bastante interessante e positivo para os elementos do grupo este trabalho de tradução. Foi possível também perceber como de facto funcionam os compiladores das linguagens mais complexas que usamos no nosso dia a dia e nem questionamos bem como funcionam.