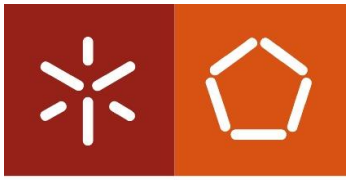

Universidade do Minho

Departamento de Informática

Processamento de Linguagens

TP2 - GIC/GT + Compiladores - Grupo 71

Leonardo Marreiros (A89537)
Pedro Fernandes (A89574)

30 de maio de 2021

Resumo

O segundo trabalho no âmbito da unidade curricular Processamento de Linguagens consistiu na elaboração de um compilador capaz de gerar código para uma máquina de stack virtual, neste caso a VM, Virtual Machine. Assim o nosso compilador passa código escrito numa linguagem por nós desenvolvida, do tipo imperativa, para a linguagem máquina da VM.

Deste modo no presente relatório estarão apresentadas as decisões conceptuais e lógicas por nós tomadas durante o percurso de construção do compilador e linguagem.

Conteúdo

Capítulo 1

Introdução

A **Vm, Virtual Machine** trata de uma máquina de pilhas (*stacks*) composta duma pilha de execução, duma pilha de chamadas, duma zona de código, de duas heaps e de quatro registos. Cada endereço desta máquina pode apontar para informação de quatros tipos: código, pilha, bloco estruturado ou string. A máquina funciona com instruções de tipo *Assembly* a partir das quais é possível gerar programas de todo o tipo equivalentes àqueles que seriam escritos noutras linguagens.

Desta forma, pretende-se desenvolver um compilador que a partir de uma linguagem imperativa à escolha gere pseudo-código, *Assembly* da Máquina Virtual VM.

Com este trabalho conseguimos colocar em prática a capacidade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT) bem como utilizar estas gramáticas em geradores de compiladores e analisadores léxicos como o Yacc e o Lex, respetivamente.

Em primeiro lugar iremos analisar o **problema**, determinar que funcionalidades implementar e identificar os desafios que serão necessários ultrapassar para implementar essas funcionalidades.

De seguida iremos apresentar a nossa **Gramática**, **solução implementada** e finalmente **alternativas, decisões e problemas de implementação**.

Para terminar iremos apresentar o resultado de alguns **testes** e por fim alguma reflexão crítica e análise do trabalho realizado na **conclusão**.

Capítulo 2

Problema

Para atingir os resultados esperados é necessário escrever uma gramática independente do contexto para o reconhecimento da linguagem definida e o respetivo analisador léxico para criar um programa que processe esta linguagem e construa as equivalentes instruções da máquina através do Yacc, com recurso a uma gramática tradutora

Em relação à linguagem definida, esta tem que cumprir os seguintes requisitos:

- **Declarar** variáveis atómicas do tipo inteiro, com as 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. No nosso caso, deve ser implementado o ciclo **for-do**.
- Declarar e manusear variáveis estruturadas do tipo ***array*** (a 1 ou 2 dimensões) de inteiros.

Capítulo 3

Gramática

3.1 Símbolos Terminais

No decorrer do trabalho foram declarados diversos símbolos terminais. No *python* estes acabaram por ser subdivididos em 3 grupos diferentes, os *tokens*, os *literals* e por fim os *reserved*. Foram definidos os seguintes ***tokens*** :

- ***INT*** : encontra todos os inteiros da seguinte forma:

  ```
  t_INT = r'(\+|-)?(\d+)'
  ```

- ***VAR*** : encontra todas os possíveis nomes para variáveis da seguinte forma:

  ```
  t_VAR = r'[a-zA-Z]\w*'
  ```

- ***STRING*** : encontra todas as strings da seguinte forma:

  ```
  t_STRING = r'"[^"]*"'
  ```

- ***ID*** : serve de apoio para as palavras reservadas e está definido da seguinte forma:

  ```
  def t_ID(t):
      r'if|int|scan|print|for'
      t.type = reserved.get(t.value,'ID')    # Check for reserved words
      return t
  ```

No que toca aos ***literals*** estes são todos os símbolos singulares que usamos no nosso programa com diversos usos. Eles estão definidos da seguinte forma:

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

Finalmente foram ainda usados algumas palavras reservadas (***reserved***) para reconhecer diversas palavras necessárias para a realizam das tarefas propostas :

- ***INTID*** : reconhece a palavra reservada "int"usada para inicializar variáveis, reconhecido da seguinte forma:

  ```
  reserved = {
      'if' : 'IFID',
  ```

- ***READ*** : reconhece a palavra reservada "scan"usada para ler do *stdin*, reconhecido da seguinte forma:

  ```
      'scan' : 'READ',
  ```

- ***WRITE*** : reconhece a palavra reservada "print"usada para escrever no *stdout*, reconhecido da seguinte forma:

  ```
      'print' : 'WRITE',
  ```

- ***IFID*** : reconhece a palavra reservada "if"usada para realizar *if statments*, reconhecido da seguinte forma:

  ```
      'if' : 'IFID',
  ```

- ***FORID*** : reconhece a palavra reservada "for"usada para realizar *for loops*, reconhecido da seguinte forma:

  ```
      'for' : 'FORID'
  ```

3.2 Símbolos Não Terminais

Para além dos símbolos terminais, são necessários também alguns símbolos não terminais para a construção de uma gramática. Desta forma, para podermos realizar todas as operações pretendidas no programa foi necessário a construção de uma gramática apropriada.

3.2.1 Aritmética

Para a realização de funções aritméticas é necessário ter em consideração as prioridades das operações. Para tal foram necessários o *Expr*, o *Termo* e o *Factor*.

```
Expr -> Expr '+' Termo
      | Expr '-' Termo
      | Termo

Termo -> Termo '*' Factor
       | Termo '/' Factor
       | Termo '%' Factor
       | Factor

Factor -> '(' Expr ')'
        | INT
        | VAR
        | VAR '[' INT ']'
        | VAR '[' INT ']' '[' INT ']'
```

3.2.2 Inicialização/Declarações

No nosso programa existem três tipos de variáveis que podem ser declaradas, inteiros, vectores e matrizes, daí a necessidade do símbolo não terminal *Init*. Para se poder fazer diversas inicializações seguidas criou-se o *InitBlock* que permite a criação de uma lista de inicializações.

```
InitBlock -> Init
           | InitBlock Init

Init -> INTID VAR
      | INTID '[' INT ']' VAR
      | INTID '[' INT ']' '[' INT ']' VAR
```

3.2.3 Atribuições/Instruções

Existem diversos comandos possíveis para serem executados no nosso programa, os quais estão demonstrados a partir do símbolo não terminal *Command*. Para poder haver uma lista de comandos seguidos foi utilizado o *Body* que contém um ou mais comandos. Nesta secção podemos ainda ver o *Atrib* e seus derivados que servem para atribuir valores a todo o tipo de variáveis presente na linguagem.

```
Command -> Atrib
         | Read
         | Write
         | IfStatement
         | ForStatement

Atrib -> VAR '=' Expr
       | VAR '=' Read
       | VAR '[' INT ']' '=' Expr
       | AtribArray '=' Expr
       | VAR '[' INT ']' '=' Read
       | AtribArray '=' Read
       | VAR '[' INT ']' '[' INT ']' '=' Expr
       | VAR '[' INT ']' '[' INT ']' '=' Read
       | AtribMatrix '=' Expr
       | AtribMatrix '=' Read

AtribArray -> VAR '[' VAR ']'

AtribMatrix-> VAR '[' VAR ']' '[' VAR ']'

Body -> Command
      | Body Command
```

3.2.4 Leitura/Escrita

Quanto à leitura do *stdin* foi necessário recorrer simplesmente ao *Read* porque os parâmetros para esta instrução são sempre os mesmos. No que toca à escrita foi usado o *Write* que tem diversas opções, uma cara cada tipo de variável que se pode ler, quer serja esta uma string, um simples inteiro, ou até um elemento de um *array* ou matriz.

```
Read -> READ '(' ')'

Write -> WRITE '(' STRING ')'
       | WRITE '(' VAR ')'
       | WRITE '(' VAR '[' INT ']' ')'
       | WRITE '(' AtribArray ')'
       | WRITE '(' VAR '[' INT ']' '[' INT ']' ')'
       | WRITE '(' AtribMatrix ')'
```

3.2.5 Condicionais

De modo a poderem ser usadas condições na nossa linguagem, foi necessária a inclusão do símbolo não terminal *Cond*. Desta forma é possível fazer todas comparações possíveis, desde igual e diferente até maior, maior ou igual, menor e menor ou igual.

```
Cond -> Factor '=' '=' Factor
      | Factor '!' '=' Factor
      | Factor '<' '=' Factor
      | Factor '>' '=' Factor
      | Factor '>' Factor
      | Factor '<' Factor
```

3.2.6 Instrução If

Para a realização de condições foi adiciona o *IFStatement* que representa o corpo completo desta instrução, enquanto que o *If* representa o cabeçalho do mesmo.

```
IfStatement -> If '{' Body '}'

If -> IFID '(' Cond ')'
```

3.2.7 Ciclo For

De modo a realizar o ciclo correspondente ao número do nosso grupo foi necessário a criação de diversos símbolos não terminais de modo a ser possível gerar todas as instruções máquina fundamentais para realizar este tipo de ciclos.

```
ForStatement -> For '{' Body '}'

For -> FORID '(' Atrib InitLabel ForCond ')'

InitLabel -> ';'

ForCond -> Cond
```

Capítulo 4

Solução

4.1 Preliminares e Estruturas de Dados

Um programa é definido por um conjunto de declarações seguido de um conjunto de instruções que podem ser de vários tipos. Para isso começamos por definir as seguintes regras a partir das quais o resto do programa é reconhecido:

```
def p_Prog(p):
    "Prog : InitBlock"


def p_Prog_Complete(p):
    "Prog : InstrBlock"
```

Foram necessárias algumas varáveis globais que irão ser referênciadas nos próximos tópicos:

- `errorCounter`: criado para contar o número de erros originados por certas instruções ilegais na nossa linguagem.
- `condCounter`: criado para contar o número de instruções condicionais no programa. Utilizado em conjunto com o `condStack`.
- `condStack`: criado para auxiliar com a numeração das *labels* referentes a ciclos. Como o nome indica, funciona como uma stack onde são efetuadas operações de *push* e *pop*.
- `output`: *string* à qual é concatenada instruções da máquina virtual à medida que o programa é reconhecido.
- `parser.register`: dicionário onde são guardadas as váriáveis e os seus endereços. Cada variável no dicionário tem como valor o triplo (endereço, tamanho horizontal, tamanho vertical). Estes dois últimos valores são úteis especialmente na criação de *arrays* de uma e duas dimensões respetivamente.
- `parser.counter`: variável que guarda o número de declarações efetuadas.

4.2 Aritmética

Consultando a parte da gramática referente a este aspeto, podemos verificar que é composta por três regras principais: `Expr`, `Termo` e `Factor`.

4.2.1 Factor

Começando pelo `Factor`, estas regras constituem a base do problema, é aqui que são identificadas as instruções `pushi` e `pushg`. Esta regra divide-se em 5 outras, uma para cada átomo de operação,

isto é, inteiro, variável, elemento de array com uma dimensão e elemento de array com duas dimensões. É ainda uma regra que permite a recursividade das operações acima.

```
def p_Factor_INT(p):
    "Factor : INT"
```

De forma simples, quando a regra `Factor` reconhece um **inteiro**, concatena a string `pushi` junto com o número identificado.

```
def p_Factor_VAR(p):
    "Factor : VAR"
```

Para uma **variável**, concatena a string `pushg` junto com o endereço da variável identificada, endereço esse que vai buscar ao `parser.register`.

```
def p_Factor_VAR_Int(p):
    "Factor : VAR '[' INT ']'"
```

Para o elemento de um **array** uni-dimensional, encontra o endereço do elemento pretendido somando a posição inicial do array (primeiro elemento do triplo correspondente à variável identificada) com o índice do array reconhecido pela regra.

```
def p_Factor_VAR_Int_Int(p):
   "Factor : VAR '[' INT ']' '[' INT ']'"
```

Finalmente, quando reconhece um elemento de um **array** bi-dimensional, para encontrar o endereço do elemento pretendido, é calculado o resultado da expressão: $v + M * i + j$ onde v corresponde ao endereço de inicio do array, M é a dimensão vertical do array, encontrada no dicionário de registos no terceiro valor do triplo correspondente à variável identificada, i corresponde ao índice horizontal reconhecido e j corresponde ao índice vertical reconhecido.

4.2.2 Termo

As regras de tradução `Termo` correspondem às operações de maior prioridade na aritmética, isto é: multiplicação, divisão e resto da divisão. Um `Termo` pode também ser um `Factor`.

Quando é identificada uma destas operações, é concatenada à string `output` a instrução VM correspondente. Por exemplo, para a multiplicação:

```
def p_Term_mull(p):
    "Term : Term '*' Factor"
    global output
    if (error != 1):
        output += ("mul\n")
```

4.2.3 Expr

Para finalizar, as regras de tradução `Expr` correspondem às operações de menor prioridade na aritmética, isto é: adição e subtração. Uma `Expr` pode também ser um `Termo`.

Quando é identificada uma destas operações, é concatenada à string `output` a instrução VM correspondente. Por exemplo, para a adição:

```
def p_Expr_add(p):
    "Expr : Expr '+' Term"
    global output
    if (error != 1):
        output += ("add\n")
```

É importante notar que uma das regras do `Fator` inclui um `Expr`:

```
def p_Factor_group(p):
    "Factor : '(' Expr ')'"
```

Isto significa que podem ser realizadas várias operações aninhadas.

4.3 Inicializações/Declarações

Como é normal, são permitidas declarações ilimitadas logo, começamos por definir as seguintes regras:

```
def p_InitBlock_Single(p):
    "InitBlock : Init"


def p_InitBlock_List(p):
    "InitBlock : InitBlock Init"
```

Existem três diferentes tipos de inicializações: inicializar uma variável, inicializar um array com uma dimensão e inicializar um array com duas dimensões. Todas as inicializações são feitas com o valor '0'.

```
def p_Init_Int(p):
    "Init : INTID VAR"
```

Para a inicialização da variável, é reconhecida primeiro a palavra reservada 'int' e depois a sua designação pretendida. De seguida, caso a variável não exista no `parser.registers`, então é concatenada à string `output` a instrução "pushi 0", inserida a variável no dicionário com o número atual de `parser.counter`, dimensão horizontal 1 e dimensão vertical 1. Por fim é aumentado o valor de `parser.counter`.

```
def p_Init_Array(p):
    "Init : INTID '[' INT ']' VAR"
```

Para a inicialização do array uni-dimensional, é reconhecida novamente a palavra reservada 'int', juntamente com a dimensão pretendida para o array e finalmente a sua designação. Tal como na regra anterior, caso a variável não exista no `parser.registers`, entao é concatenada à string `output` a instrução "pushn " juntamente com a dimensão reconhecida. É inserida a variável no dicionário com o número atual de `parser.counter`, dimensão horizontal igual à dimensão reconhecida (p(N)) e dimensão vertical 1. Finalmente é também aumentado o valor de `parser.counter` em N unidades.

```
def p_Init_Matrix(p):
    "Init : INTID '[' INT ']' '[' INT ']' VAR"
```

Finalmente, para a inicialização do array bi-dimensional, é reconhecida uma vez mais a palavra reservada 'int', juntamente com a dimensão horizontal e vertical para o array e finalmente a sua designação. Mais uma vez, caso a variável não exista no `parser.registers`, entao é concatenada à string `output` a instrução "pushn "juntamente com o resultado da múltiplicação da dimensão horizontal com a dimensão vertical ((p[N)*(p[M])), cujo resultado é a dimensão do array. É inserida a variável no dicionário com o número atual de `parser.counter`, dimensão horizontal e vertical igual às dimensões reconhecidas (N, e M respetivamente). É também aumentado o valor de `parser.counter` em N*M unidades.

4.4 Instruções

À semelhança das inicializações, também com as instruções não há um número limitado para as mesmas. Para isto, foram definidas as seguintes regras:

```
def p_InstrBlock_Single(p):
    "InstrBlock : Command"
```

```
def p_InstrBlock_List(p):
    "InstrBlock : InstrBlock Command"
```

4.4.1 Command

Um `Command` engloba as várias operações que se podem encontrar num programa: atribuições, leituras, escritas, condições e ciclos. A regra de tradução foi definida da seguinte forma:

```
def p_Command_Atrib(p):
    "Command : Atrib"


def p_Command_Read(p):
    "Command : Read"


def p_Command_Write(p):
    "Command : Write"


def p_Command_IfStatement(p):
    "Command : IfStatement"


def p_Command_ForStatement(p):
    "Command : ForStatement"
```

4.5 Atribuições

Esta funcionalidade é das mais importantes uma vez que a maioria das instruções feitas num programa são atribuições. Existem vários tipos de atribuições: atribuir o resultado de uma expressão a uma variável, atribuir o valor de *input* a uma variável, atribuir o resultado de uma expressão a um elemento de um array com índice inteiro, atribuir o valor de *input* a um elemento de um array com índice inteiro, atribuir o resultado de uma expressão a um elemento de um array com índice variável, atribuir o valor de *input* a um elemento de um array com índice variável. Lembrando uma vez mais que os array podem ser uni ou bi-dimensionais.

```
def p_Atrib_Expr(p):
   "Atrib : VAR '=' Expr"
```

Começando pelo tipo de atribuição mais simples, onde o resultado de uma expressão é guardado numa variável: caso a variável reconhecida esteja presente no dicionário `parse.registers`, concatena a instrução `storeg` junto com o endereço da variável inserida (valor presente no dicionário) à string global `output`.

```
def p_Atrib_Read(p):
   "Atrib : VAR  '=' Read"
```

Ao invés de atribuir um valor fixo a uma variável, é possível também atribuir o resultado de uma operação de leitura a uma variável. Para isto, são feitas as mesmas operações que no caso anterior com a adição da instrução `atoi` que converte o valor inserido no *read* para inteiro.

```
def p_Atrib_Array_Int(p):
   "Atrib : VAR '[' INT ']' '=' Expr"
```

Passando agora para as operações sobre arrays, podemos ter, por exemplo v[1] ou v[i], no primeiro caso: caso a variável exista no dicionário, é somado o valor de início do array presente em `parser.register` com o valor inteiro reconhecido referente ao índice do array. Este resultado

corresponde ao endereço no qual o elemento pretendido se encontra. Com isto, é concatenado à string de resultados a instrução `storeg` mais o resultado obtido anteriormente.

```
def p_Atrib_Array_Read_Int(p):
    "Atrib :  VAR '[' INT ']' '=' Read"
```

Quanto à atribuição utilizando a leitura, as operações são as mesmas do caso anterior com a adição da instrução `atoi` que converte o valor inserido no *read* para inteiro.

```
def p_Atrib_Array_Var(p):
    "Atrib : AtribArray '=' Expr"

def p_AtribArray(p):
    "AtribArray : VAR '[' VAR ']'"
```

Para o tipo de operação com acesso a índices de tipo variável de arrays (v[i], por exemplo), foi necessária a criação de uma regra auxiliar `AtribArray`. Esta regra reconhece o lado esquerdo da operação de atribuição e gera um conjunto importante de instruções VM necessárias a atualizar o valor pretendido. Começa por adicionar uma instrução `pushgp` que empilha o valor do *global pointer*, depois faz um `pushi` do endereço onde começa o array reconhecido, `pushg` do índice (variável) ao qual prentende aceder, e adiciona estes valores com a instrução `add`. O resto da atribuição entra na regra `p_Atrib_Array_Var(p)` onde é apenas adicionada a instrução `storen` que vai guardar o resultado da expressão calculada no índice do array reconhecido.

```
def p_Atrib_Array_Read_Var(p):
   "Atrib :  AtribArray '=' Read"
```

Quando é reconhecida uma leitura em vez de uma expressão, é utilizada a regra tradutora acima que tem um procedimento equivalente ao caso anterior com adição da instrução `atoi` para converter para inteiro o valor inserido.

```
def p_Atrib_Matrix_Int(p):
    "Atrib : VAR '[' INT ']' '[' INT ']' '=' Expr"
```

Passando agora para as atribuições sobre array bi-dimensionais, os procedimentos são maioritariamente equivalentes aos anteriores, com exceção do método de cálculo de endereços, como seria de imaginar. Equivalente à regra `p_Atrib_Array_Int(p)` no entanto o valor do endereço em `storeg` resulta da fórmula usada anteriormente $v + M * i + j$ onde v corresponde ao endereço de inicio do array, M é a dimensão vertical do array, encontrada no dicionário de registos no terceiro valor do triplo correspondente à variável identificada, i corresponde ao índice horizontal reconhecido e j corresponde ao índice vertical reconhecido.

```
def p_Atrib_Matrix_Read_Int(p):
    "Atrib : VAR '[' INT ']' '[' INT ']' '=' Read"
```

Equivalente à regra `p_Atrib_Array_Read_Int(p)` novamente com a exceção do valor do argumento em `storeg`, calculado a partir da fórmula $v + M * i + j$.

```
def p_AtribMatrix(p):
    "AtribMatrix : VAR '[' VAR ']' '[' VAR ']'"

def p_Atrib_Matrix_Var(p):
    "Atrib : AtribMatrix '=' Expr"
```

À semelhança daquilo que aconteceu com os arrays de uma dimensão, também nestes foi necessário criar uma regra adicional que reconhece a parte esquerda da atribuição, `AtribMatrix`. Desta vez, em vez de calcular a fórmula $v + M * i + j$ diretamente e colocar esse valor concatenado à string da instrução, devido à forma como este tipo de acessos funcionam em instruções da máquina virtual, tivemos de calcular esta fórmula em instruções VM, dando origem ao conjunto de instruções:

```
output += ("pushgp\n")
output += ("pushi " + str(p.parser.register.get(p[1])[0]) + "\n")
```

```
output += ("pushi " + str(p.parser.register.get(p[1])[2]) + "\n")
output += ("pushg " + str(p.parser.register.get(p[3])[0]) + "\n")
output += ("mul\n")
output += ("pushg " + str(p.parser.register.get(p[6])[0]) + "\n")
output += ("add\n")
output += ("add\n")
```

A regra `p_Atrib_Matrix_Var` adiciona a instrução `storen` após o reconhecimento da atribuição.

```
def p_Atrib_Matrix_Read_Var(p):
    "Atrib :  AtribMatrix '=' Read"
```

Semelhante à regra `p_Atrib_Array_Read_Var(p)`.

4.6 Leitura

A operação de leitura no contexto de um programa nunca é usada sozinha, isto é, as leituras que são feitas usualmente tem como objetivo guardar uma variável com o *input* inserido, por exemplo. Deste modo, apesar de poder ser usada sozinha, a sua utlização irá aparecer mais no contexto de atribuições (já referidas). A regra tradutora apenas junta à string global a instrução `read`.

```
def p_Read(p):
    "Read : READ '(' ')'"
    global output
    output += ("read \n")
```

4.7 Escrita

Não só é possível escrever uma string, também é possível escrever uma variável e um elemento de um array de uma ou duas dimensões dado um índice inteiro ou uma variável.

```
def p_Write_String(p):
    "Write : WRITE '(' STRING ')'"
```

Regra de tradução que reconhece uma instrução de escrita de uma string. Apenas faz `pushs` da string inserida e de seguida `writes`.

```
def p_Write_Var(p):
    "Write : WRITE '(' VAR ')'"
```

Para escrever uma variável, é preciso aceder primeiro à entrada do dicionário referente à variável reconhecida para encontrar o endereço onde se encontra. Depois, junta-se este valor à instrução `pushg` seguido de `stri` que passa um valor de tipo inteiro para uma string e finalmente `writes` da string resultante que corresponde ao valor pretendido.

```
def p_Write_Array_Int(p):
    "Write : WRITE '(' VAR '[' INT ']' ')'"
```

Para escrever o elemento de um array, é semelhante ao caso anterior apenas havendo diferenças no cálculo do endereço para a instrução `pushg`. Neste caso, à semelhança de casos anteriores, temos de somar ao endereço de início do array o valor do índice reconhecido (p[5]). As restantes instruções (`stri`, `writes`) são iguais.

```
def p_Write_Array_Var(p):
   "Write : WRITE '(' AtribArray ')'"
```

No caso de reconhecer a escrita de um elemento de um array dado um índice variável, fizemos uso novamente da regra `AtribArray` e seguimos este reconhecimento com as instruções: `loadn`, para aceder ao índice pretendido, `stri` para fazer a conversão deste valor para string e `writes` para escrever este resultado.

```
def p_Write_Matrix_Int(p):
    "Write : WRITE '(' VAR '[' INT ']' '[' INT ']' ')'"
```

No caso de arrays de duas dimensões, as instruções são semelhantes às originadas nos arrays com uma dimensão: `pushg` concatenado ao resultado da fórmula já usada para determinar o endereço na máquina virtual do elemento pretendido no caso de arrays bi-dimensionais; `stri` para fazer a conversão deste valor para string e `writes` para escrever este resultado.

```
def p_Write_Matrix_Var(p):
    "Write : WRITE '(' AtribMatrix ')'"
```

Em concordância com aquilo que foi feito nos arrays com uma dimensão, também neste caso utilizamos a regra usada anteriormente `AtribMatrix`. De seguida geramos as instruções `loadn`, `stri` e `writes` para escrever este resultado.

4.8 Condicionais

A máquina virtual VM contém instruções de comparação de inteiros pelo que a implementação do reconhecimento e geração de código Assembly foi bastante simples consistindo apenas na adição da instrução correspondente ao tipo de condição. Por exemplo:

```
def p_Cond_Equals(p):
    "Cond : Factor '=' '=' Factor"
    global output
    output += ("equal\n")


def p_Cond_NotEquals(p):
    "Cond : Factor '!' '=' Factor"
    global output
    output += ("equal\nnot\n")
```

4.9 Instrução If

```
def p_IfStatement(p):
    "IfStatement : If '{' Body '}'"
    global output
    output += ("endif" + str(condStack.pop()) + ":" + "\n")

def p_If(p):
    "If : IFID '(' Cond ')'"
    global condCounter, condStack, output
    condStack.append(condCounter)
    output += ("jz endif" + str(condCounter) + "\n")
    condCounter += 1
```

A instrução If foi dividida em duas regras. Primeiro é reconhecida a condição if com `p_If(p)` que consiste na palavra reservada 'if' e de uma condição cujas regras já foram analisadas. Nesta fase, é adicionado à *stack* `condStack` o valor de `condCounter` e adicionada a instrução de salto condicional com a respetiva numeração à string de resultados. A regra `p_IfStatement` corresponde a um bloco If na linguagem de programação. Ora, como é habitual, o corpo deste bloco é constituido por um número ilimitado de instruções. Deste modo, faz parte regra de tradução desta funcionalidade um `Body` que é definido da seguinte forma:

```
def p_Body_Single(p):
    "Body : Command"
```

```
def p_Body_List(p):
    "Body : Body Command"
```

Assim, quando o compilador chega ao último '}' reconhece esse pedaço de código como um `IfStatement` e gera a instrução referente à label para a qual se o valor da condição for nulo então é atribuído ao registo *pc* o seu endereço. A numeração correspondente ao ciclo do qual vai sair corresponde ao valor à cabeça da `coondStack` pelo que é feito um *pop* e é concatenado este valor à label.

4.10 Ciclo For

```
def p_ForStatement(p):
    "ForStatement : For '{' Body '}'"
    global output
    output += ("jump repeat" + str(condStack[(len(condStack)-1)]) + "\n")
    output += ("repeatend" + str(condStack.pop()) + ":\n")

def p_For(p):
    "For : FORID '(' Atrib InitLabel ForCond ')'"

def p_InitLabel(p):
    "InitLabel : ';'"
    global condCounter, condStack, output
    condStack.append(condCounter)
    output += ("repeat" + str(condCounter) + ":\n")
    condCounter += 1

def p_ForCond(p):
    "ForCond : Cond"
    global output
    output += ("jz repeatend" + str(condStack[(len(condStack)-1)]) + "\n")
```

Um ciclo for está dividido em várias regras: `ForStatement` é a regra mais geral e reconhece o ciclo na sua totalidade. É constituido por um `Body` que já foi apresentado e uma regra `For`. Esta regra `For` por sua vez é identificada com a palavra reservada 'for' seguida de uma Atribuição, uma `InitLabel` e uma `ForCond`. O `InitLabel` é o que nos permite colocar a `label` de início de ciclo. De seguida o `ForCond` permite-nos fazer o teste da condição do ciclo, para ver se é suposto quebrar ou não o mesmo. Tal como no `If` foi necessário recorrer ao `condCounter` e ao `condStack` de modo a permitir estabelecer a numeração de cada uma destas `labels` e instruções de salto. É necessário realçar que para funcionar da melhor forma é necessário a cada instrução `"repeat"` realizar um `push` à `condStack` uma vez que representa o início da instrução e a cada `"repeatend"` é necessário proceder ao seu `pop`. No caso das instruções intermédias que interagem com `labels`, a `"repeatend"` e a `"jz repeatend"` é necessário ir à `condStack` buscar o elemento que se encontra no índice equivalente ao comprimento da mesma subtraído de 1.

4.11 Erros

Ao longo da implementação deste compilador e da definição da nossa linguagem de programação, fizemos questão de estabelecer cenários de erro. Sempre que uma operação ilegal como aceder a uma variável que não existe ou tentar somar um array com um inteiro é reconhecida, é gerada uma instrução de erro na linguagem VM. Os diferentes cenário de erro identificados foram:

- Em atribuições onde são utilizadas variáveis, caso a variável não exista no dicionário `parser.register`, é gerado o erro `"variable <variable name> does not exist"`.
- No caso de se tentar somar um array com um inteiro, é gerado o erro `"Unsupported operand type(s)"`.

- Caso se tente indexar uma variável de tipo inteiro, é gerado o erro `"'int' object is not subscriptable"`.
- Caso se tente indexar uma variável do tipo array fora dos seus limites, é gerado o erro `"Segmentation fault!  variable <variable name>"`
- Caso se tente inicializar uma variável que já exista anteriormente, é gerado o erro `"variable <varible name> already exists"`

Sempre que um erro é identificado, o contador `errorCounter` é incrementado. Quando o compilador está pronto para escrever a string resultante com todas as instruções para o ficheiro de *output*, primeiro verifica o contador de erros. Na eventualidade deste ser maior que 0, são filtrados todas as instruções de erro e apenas escritas estas instruções.

Capítulo 5

Alternativas, Decisões e Problemas de Implementação

Aquando do reconhecimento das instruções If e ciclos For, apercebemo-nos que talvez seria melhor efetuar a leitura do ficheiro todo em vez de o ler linha a linha pelo que esta é uma das principais alternativas que podiamos ter implementado. O facto da leitura estar a ser feita linha a linha, dificultou o processo de reconhecimento destas instruções na medida em que estas englobam várias linhas. A solução encontrada foi identificar as linhas em que estavam presentes '{' e ir construindo uma string com todas as instruções presentes no corpo dessa operação até '}'. Assim, conseguimos construir a operação como se de uma única linha se tratasse, o que possibilitou o seu reconhecimento.

Consideramos que a implementação de uma stack para auxiliar na numeração das *labels* de intruções If e ciclos For foi uma decisão bastante inteligente, na medida em que fornecia exatamente aquilo que estavamos à procura. Sempre que começa um ciclo é feito *push* do contador de ciclos, e quando acaba é feito um *pop*.

Além disso, por uma questão de simplificação, decidimos não colocar o incremento do ciclo For no seu "*header*", pelo que este incremento tem de ser escrito no código dentro do ciclo.

Finalmente, a implementação de uma inicialização de um array com uma variável (int [N] v, por exemplo) revelou-se ser impossível na medida em que estamos limitados às intruções presentes na maquina virtual.

Capítulo 6

Testes realizados e Resultados

Nesta secção podemos ver diversos programas escritos na linguagem por nós criada, que após ser lida pelo compilador, vai gerar código máquina para que a VM o possa correr. Assim para demonstrar o bom funcionamento do nosso compilador decidimos não só realizar os cinco programas sugeridos pelos docentes, mas também um outro extra (ex 6), que demonstra o funcionamento de matrizes, ou seja, *arrays* de duas dimensões.

6.1 Ex 1

O Exercício 1 pedia que fossem lidos 4 números e o programa tinha de dizer, se esses 4 números poderiam ser lados de um quadrado. Para isso é necessário verificar se todos os números inseridos são iguais. De seguida encontra-se o código escrito para tal efeito :

```
int a
int b
int c
int d
int e
a = scan()
b = scan()
c = scan()
d = scan()
if(a == b){
    if(b==c){
        if(c==d){
            e = 1
        }
    }
}
if(e==1){
    print("Quadrado!")
}
if(e==0){
    print("Nao Quadrado!")
}
```

Caso o input sejam 4 números iguais como no exemplo seguinte, o programa irá imprimir "Quadrado!"

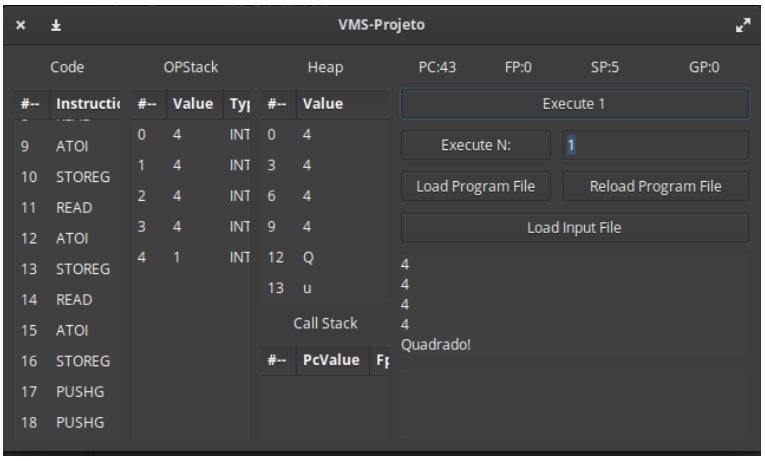

Figura 6.1: Ex 1 - resultado input(4,4,4,4)

Caso no input exista pelo menos um lado com dimensão diferente como no exemplo seguinte, o programa irá imprimir "Nao Quadrado!"

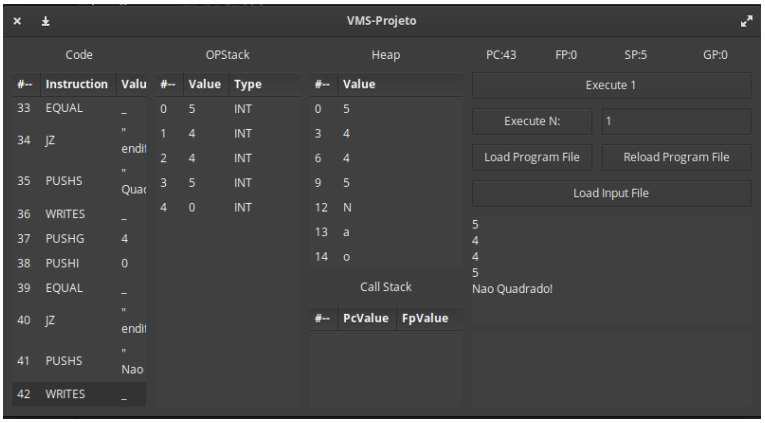

Figura 6.2: Ex 1 - resultado input(5,4,4,5)

6.2 Ex 2

O Exercício 2 pedia que o programa lê-se um inteiro N, de seguida teria de ler N inteiros e por fim dizer qual o menor deles. Desta forma é necessário simplesmente necessário guardar o inteiro N e armazenar o inteiro mais pequeno até ao momento. É de notar que o nosso compilador aceita inteiros negativos.

```
int n
n = scan()
int i
int res
int read
res = scan()
n = n - 1
for(i = 0 ; i < n){
    read = scan()
    if(read < res){
        res = read
    }
    i = i + 1
}
print("O menor numero inserido e: ")
print(res)
```

Para a realização do teste seguinte considerou-se o N = 4, ou seja serão inseridos 4 inteiros. De seguida foram inseridos os valores 56, 0, -5 e 79. O programa deu como resultado o menor número, ou seja -5.

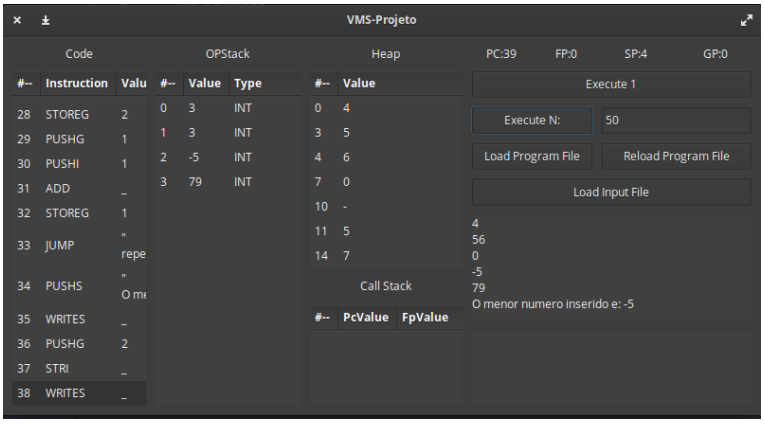

Figura 6.3: Ex 2 - resultado

6.3 Ex 3

O Exercício 3 pedia que a partir de um número N inserido pelo utilizador, fossem de seguida lidos N números e de seguida calculado o seu produtório, ou seja, caso fossem inseridos os valores 1,2,3,4 o programa iria imprimir o resultado = 1 * 2 * 3 * 4.

```
int n
n = scan()
int i
int res
int read
res = scan()
n = n - 1
for(i = 0 ; i < n){
    read = scan()
    res = read * res
    i = i + 1
}
print("O produto dos numeros inseridos e: ")
print(res)
```

Para o primeiro teste foram adicionados 4 inteiros (N=4). Os números inseridos foram 2, 3, 3, 2 cujo produtório é 36 como podemos ver na seguinte imagem:

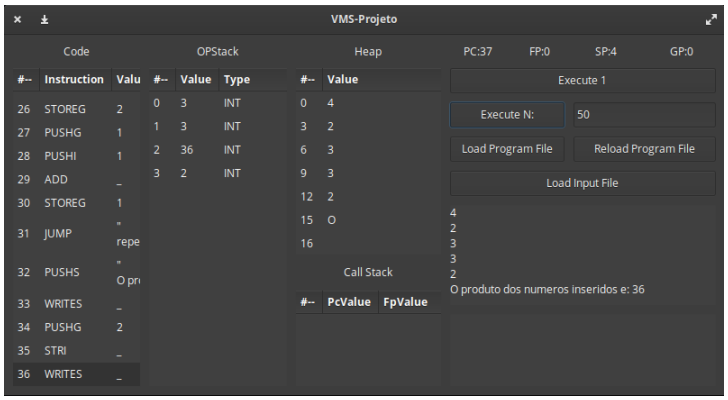

Figura 6.4: Ex 3 - resultado positivos

Para o segundo teste foram adicionados também 4 inteiros (N=4), no inteiro foi adiciona um número negativo. Os inteiros inseridos foram portanto -5, 4, 2 e 1 cujo produtório é -40 como indicado de seguida:

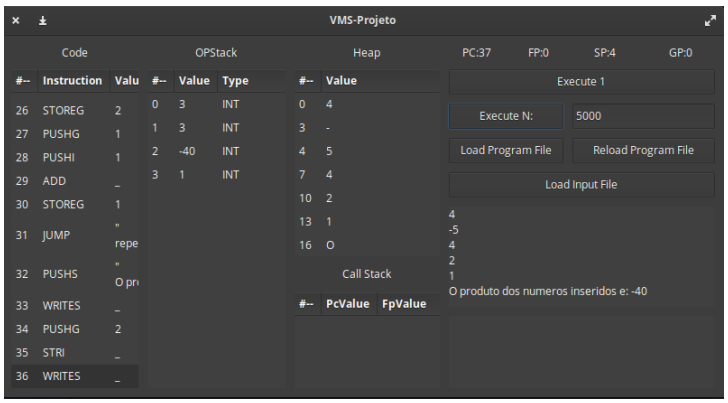

Figura 6.5: Ex 3 - resultado positivos

6.4 Ex 4

O Exercício 4 pedia para que o programa fosse lendo números naturais do stdin e cada vez que aparecia um número ímpar, esse número deveria ser impresso e no final deveria aparecer também no stdout o número total de ímpares inseridos. Note que consideramos que para parar terminar o programa decidimos que este termina caso o utilizador insira -1.

```
int i
int count
count = 0
int res
print("Insira numeros (termina com -1)\n")
for(i = 0 ; i != -1){
    i = scan()
    if ( i != -1){
        res = i % 2
        if( res != 0 ){
            print("\nNumero impar encontrado: ")
            print(i)
            print("\n")
            count = count + 1
        }
    }
}
print("Numeros impares inseridos: ")
print(count)
```

Para o seguinte teste foram inseridos por esta ordem os números 3, 1, 2, 6, 9 e por fim -1 para terminar a inserção. Como podemos ver em seguida, para os números ímpares (3, 1, 9) eles foram impressos como tendo sido um natural ímpar encontrado, e no final aparece, como pretendido, que foram encontrados 3 ímpares.

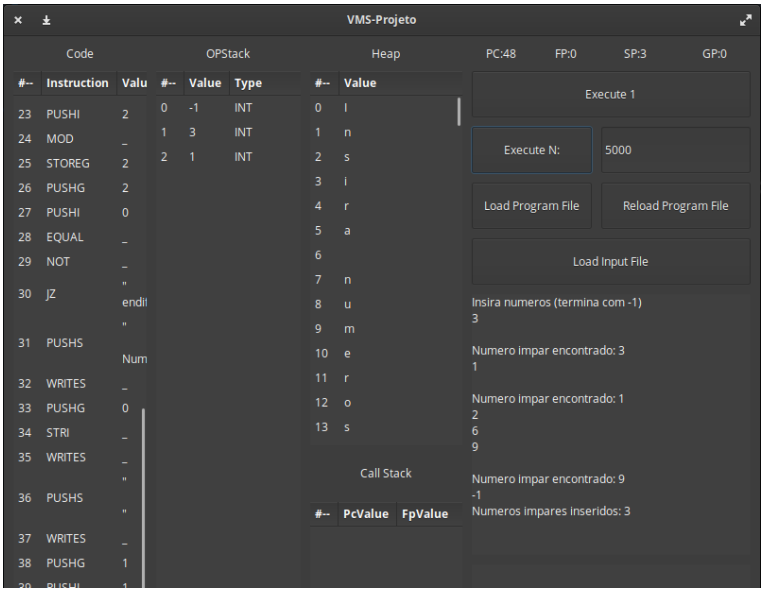

Figura 6.6: Ex 4 - resultado

6.5 Ex 5

O Exercício 5 tinha proposto ler e armazenar N números (sendo N um inteiro do programa) num *array* e de seguida imprimir esses mesmos números na ordem inversa pela qual foram inseridos. Para tal foi escrito o seguinte código:

```
int i
int [5] n
for(i=0; i < 5){
    print("Insira um numero!\n")
    n[i] = scan()
    i = i + 1
}
print("Array por ordem inversa: ")
for(i = 4; i >= 0){
    print(" ")
    print(n[i])
    i = i - 1
}
```

Sendo 5 o inteiro N do nosso programa, ou seja, o tamanho do *array* pretendido, são lidos 5 números do *stdin*. No seguinte exemplo são lidos e colocados no *array* os números pela seguinte ordem (1,2,3,4,5). No fim o programa imprime o este *array* pela ordem inversa como pedido como se pode observar de seguida:

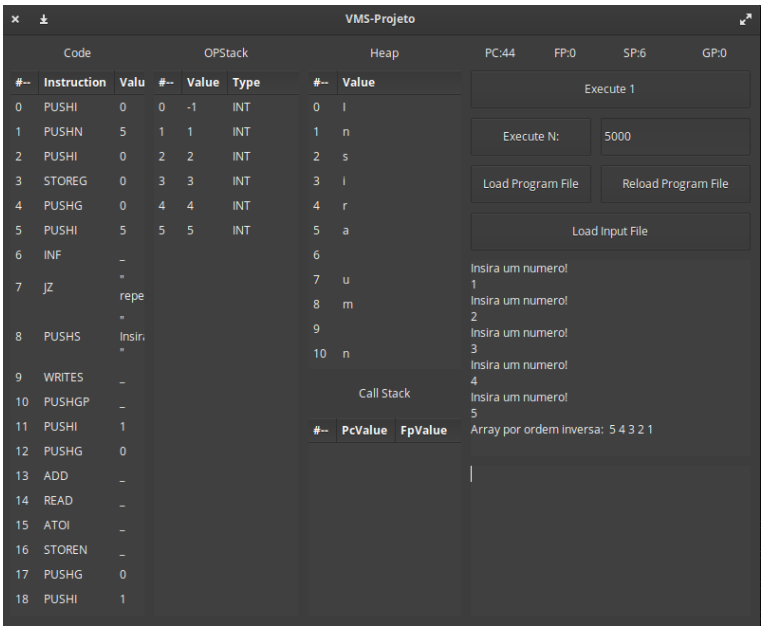

Figura 6.7: Ex 5 - resultado

6.6 Ex 6

O Exercício 6 tem como objetivo demonstrar o uso de matrizes na nossa linguagem. Para isso desenvolvemos um programa que pede que o utilizador insira números para preencher uma matriz 3x2. De seguida o Programa imprime a matriz resultante do que foi inserido seguido da matriz transposta dessa mesma matriz.

```
int [3][2] v
int i
int j
print("Insira os numeros de uma matriz 3x2:\n")
for(i=0; i < 3){
    for(j=0; j < 2) {
        v[i][j] = scan()
        j = j + 1
    }
    i = i + 1
}
print("A matriz original e:\n")
for(i=0; i < 3){
    print("[")
    for(j=0; j < 2) {
        print(v[i][j])
        print(" ")
        j = j + 1
    }
    print("]\n")
    i = i + 1
}
print("\nA matriz transposta e:\n")
for(j=0; j < 2){
    print("[")
    for(i=0; i < 3) {
        print(v[i][j])
        print(" ")
        i = i + 1
    }
    print("]\n")
    j = j + 1
}
```

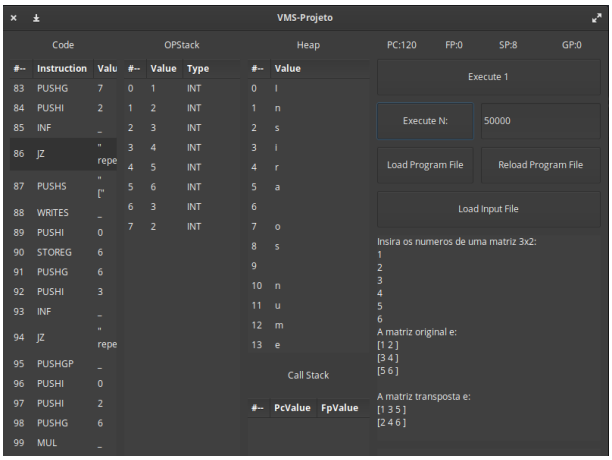

Figura 6.8: Ex 6 - resultado da inserção (1, 2, 3, 4, 5, 6)

Capítulo 7

Conclusão

Dada por concluída o trabalho, consideramos relevante efetuar uma análise crítica do trabalho realizado.

Através do desenvolvimento deste projeto conseguimos melhorar os nossos conhecimentos relativamente à capacidade de escrever gramáticas independentes de contexto e gramáticas tradutoras; conseguimos desenvolver um processadores de linguagem segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora; e colocamos em prática a utilização de geradores de compiladores baseados em gramáticas tradutoras, concretamente o Yacc, versão PLY do Python, completado pelo gerador de analisadores léxicos Lex, também versão PLY do Python.

Concluído o desenvolvimento do trabalho, notamos que existiram aspetos positivos a realçar, entre eles a preocupação com a geração e tratamento de casos de erro no código, a definição de uma linguagem fácil de entender, a geração de exemplos de código (incluindo o exemplo extra) que demostram as várias funcionalidades do projeto e a implementação de soluções inteligentes para os diversos problemas que surgiram ao longo da realização do problema.

Por outro lado, também existiram algumas dificuldades, tais como perceber quais as intruções VM para armazenar num array dado um índice de tipo variável e a numeração das *labels* e a ordem das instruções nos ciclos For. Apesar de ter sido necessário um cuidado e atenção redobrada, conseguimos superar estas dificuldades.

Com isto, podemos concluir que consideramos que houve um balanço positivo do trabalho realizado dado que as dificuldades sentidas foram superadas e foram cumpridos todos os requisitos. Foi criada uma linguagem de programação e, com base nesta, implementado um compilador capaz de reconhecer e gerar o código respetivo para uma máquina de stack virtual.

Como trabalho futuro, gostaríamos de melhorar a implementação das instruções If, adicionando a funcionalidade de reconhecer operações to tipo `and` e `or`; melhorar a implementação das condições permitindo o uso de atribuições e não só átomos de operações (Factor) e melhorar a implementação do ciclo For, acrescentando a instrução de incremento no seu "*header*"como é habitual nas linguagens de programação comuns.

Capítulo 8

Anexos

8.1 tp2_lex.py

```
import ply.lex as lex


reserved = {
   'if' : 'IFID',
   'int' : 'INTID',
   'scan' : 'READ',
   'print' : 'WRITE',
   'for' : 'FORID'
}


tokens = ['INT', 'VAR', 'STRING', 'ID'] + list(reserved.values())
literals = ['(', ')', '+', '-', '*', '/', '=', '>', '<', '!', '{', '}', ';', '%', '[', ']']

def t_ID(t):
    r'if|int|scan|print|for'
    t.type = reserved.get(t.value,'ID')    # Check for reserved words
    return t

t_INT = r'(\+|-)?(\d+)'
t_VAR = r'[a-zA-Z]\w*'

t_STRING = r'"[^"]*"'


t_ignore = " \t\n\r"

def t_error(t):
    print("Ilegal Character: ", t.value[0])
    t.lexer.skip(1)

lexer = lex.lex()
```

8.2 tp2_yacc.py

```
import sys
import ply.yacc as yacc
from tp2_lex import tokens
import re

error = 0
condCounter = 0
condStack = []
errorCounter = 0

output = ""

# ------------------------------------------------------------------------------
# Prog ------------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_Prog(p):
    "Prog : InitBlock"


def p_Prog_Complete(p):
    "Prog : InstrBlock"

# ------------------------------------------------------------------------------
# InitBlock -------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_InitBlock_Single(p):
    "InitBlock : Init"


def p_InitBlock_List(p):
    "InitBlock : InitBlock Init"

# ------------------------------------------------------------------------------
# Init ------------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_Init_Int(p):
    "Init : INTID VAR"
    global output, errorCounter
    if (p.parser.register.get(p[2]) == None):
        output += ("pushi 0\n")
        p.parser.register.update({p[2]: (p.parser.counter, 1, 1)})
        p.parser.counter += 1
    else:
        output += ("err \"variable '" + p[2] + "' already exists\"\n")
        errorCounter += 1


def p_Init_Array(p):
    "Init : INTID '[' INT ']' VAR"
    global output, errorCounter
    if (p.parser.register.get(p[5]) == None):
```

```
        output += ("pushn " + p[3] + "\n")
        p.parser.register.update({p[5]: (p.parser.counter, int(p[3]), 1)})
        p.parser.counter += int(p[3])
    else:
        output += ("err \"variable '" + p[5] + "' already exists\"\n")
        errorCounter += 1


def p_Init_Matrix(p):
    "Init : INTID '[' INT ']' '[' INT ']' VAR"
    global output, errorCounter
    if (p.parser.register.get(p[8]) == None):
        output += ("pushn " + str(int(p[3])*int(p[6])) + "\n")
        p.parser.register.update(
            {p[8]: (p.parser.counter, int(p[3]), int(p[6]))})
        p.parser.counter += int(p[3]) * int(p[6])
    else:
        output += ("err \"variable '" + p[8] + "' already exists\"\n")
        errorCounter += 1

# ------------------------------------------------------------------------
# InstrBlock -------------------------------------------------------------
# ------------------------------------------------------------------------

def p_InstrBlock_Single(p):
    "InstrBlock : Command"


def p_InstrBlock_List(p):
    "InstrBlock : InstrBlock Command"

# ------------------------------------------------------------------------
# Command ----------------------------------------------------------------
# ------------------------------------------------------------------------

def p_Command_Atrib(p):
    "Command : Atrib"


def p_Command_Read(p):
    "Command : Read"


def p_Command_Write(p):
    "Command : Write"


def p_Command_IfStatement(p):
    "Command : IfStatement"


def p_Command_ForStatement(p):
    "Command : ForStatement"

# ------------------------------------------------------------------------
# Atrib ------------------------------------------------------------------
```

```
# ------------------------------------------------------------------------------

def p_Atrib_Expr(p):
    "Atrib : VAR '=' Expr"
    global output, errorCounter
    if(p.parser.register.get(p[1]) == None):
        output += ("err \"variable '" + p[1] + "' does not exist\"\n")
        errorCounter += 1
    elif (error != 1):
        output += ("storeg " + str(p.parser.register.get(p[1])[0]) + "\n")


def p_Atrib_Read(p):
    "Atrib : VAR  '=' Read"
    global output, errorCounter
    if(p.parser.register.get(p[1]) == None):
        output += ("err \"variable '" + p[1] + "' does not exist\"\n")
        errorCounter += 1
    elif (error != 1):
        output += ("atoi \n")
        output += ("storeg " + str(p.parser.register.get(p[1])[0]) + "\n")


def p_Atrib_Array_Int(p):
    "Atrib : VAR '[' INT ']' '=' Expr"
    global output, errorCounter
    if(p.parser.register.get(p[1]) == None):
        output += ("err \"variable '" + p[1] + "' does not exist\"\n")
        errorCounter += 1
    elif (int(p[3]) >= p.parser.register.get(p[1])[1]):
        output += ("err \" Segmentation fault! variable '" + p[1] + "' \"\n")
        errorCounter += 1
    elif (error != 1):
        output += ("storeg " +
                   str(p.parser.register.get(p[1])[0] + int(p[3])) + "\n")


def p_Atrib_Array_Var(p):
    "Atrib : AtribArray '=' Expr"
    global output
    output += ("storen\n")


def p_AtribArray(p):
    "AtribArray : VAR '[' VAR ']'"
    global output, errorCounter
    if(p.parser.register.get(p[1]) == None):
        output += ("err \"variable '" + p[1] + "' does not exist\"\n")
        errorCounter += 1
    elif (error != 1):
        output += ("pushgp\n")
        output += ("pushi " + str(p.parser.register.get(p[1])[0]) + "\n")
        output += ("pushg " + str(p.parser.register.get(p[3])[0]) + "\n")
        output += ("add\n")
```

```
def p_Atrib_Array_Read_Int(p):
    "Atrib :  VAR '[' INT ']' '=' Read"
    global output, errorCounter
    if(p.parser.register.get(p[1]) == None):
        output += ("err \"variable '" + p[1] + "' does not exist\"\n")
        errorCounter += 1
    elif (int(p[3]) >= p.parser.register.get(p[1])[1]):
        output += ("err \" Segmentation fault! variable '" + p[1] + "' \"\n")
        errorCounter += 1
    elif (error != 1):
        output += ("atoi \n")
        output += ("storeg " +
                   str(p.parser.register.get(p[1])[0] + int(p[3])) + "\n")


def p_Atrib_Array_Read_Var(p):
    "Atrib :  AtribArray '=' Read"
    global output, errorCounter
    if (error != 1):
        output += ("atoi \n")
        output += ("storen\n")


def p_Atrib_Matrix_Int(p):
    "Atrib : VAR '[' INT ']' '[' INT ']' '=' Expr"
    global output, errorCounter
    var = p.parser.register.get(p[1])
    if(p.parser.register.get(p[1]) == None):
        output += ("err \"variable '" + p[1] + "' does not exist\"\n")
        errorCounter += 1
    elif (int(p[3]) >= var[1] or int(p[6]) >= var[2]):
        output += ("err \" Segmentation fault! variable '" + p[1] + "' \"\n")
        errorCounter += 1
    elif (error != 1):
        output += ("storeg " + str(var[0] + var[2] * int(p[3]) + (int(p[6]))) + "\n")


def p_Atrib_Matrix_Read_Int(p):
    "Atrib : VAR '[' INT ']' '[' INT ']' '=' Read"
    global output, errorCounter
    var = p.parser.register.get(p[1])
    if(p.parser.register.get(p[1]) == None):
        output += ("err \"variable '" + p[1] + "' does not exist\"\n")
        errorCounter += 1
    elif (int(p[3]) >= var[1] or int(p[6]) >= var[2]):
        output += ("err \" Segmentation fault! variable '" + p[1] + "' \"\n")
        errorCounter += 1
    elif (error != 1):
        output += ("atoi \n")
        output += ("storeg " + str(var[0] + var[2] * int(p[3]) + (int(p[6]))) + "\n")

def p_AtribMatrix(p):
    "AtribMatrix : VAR '[' VAR ']' '[' VAR ']'"
    global output, errorCounter
    if(p.parser.register.get(p[1]) == None):
        output += ("err \"variable '" + p[1] + "' does not exist\"\n")
```

```
            errorCounter += 1
        elif (error != 1):
            output += ("pushgp\n")
            output += ("pushi " + str(p.parser.register.get(p[1])[0]) + "\n")
            output += ("pushi " + str(p.parser.register.get(p[1])[2]) + "\n")
            output += ("pushg " + str(p.parser.register.get(p[3])[0]) + "\n")
            output += ("mul\n")
            output += ("pushg " + str(p.parser.register.get(p[6])[0]) + "\n")
            output += ("add\n")
            output += ("add\n")

def p_Atrib_Matrix_Var(p):
    "Atrib : AtribMatrix '=' Expr"
    global output
    output += ("storen\n")

def p_Atrib_Matrix_Read_Var(p):
    "Atrib :  AtribMatrix '=' Read"
    global output, errorCounter
    if (error != 1):
        output += ("atoi \n")
        output += ("storen\n")

# ------------------------------------------------------------------------------
# Read ------------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_Read(p):
    "Read : READ '(' ')'"
    global output
    output += ("read \n")

# ------------------------------------------------------------------------------
# Write -----------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_Write_String(p):
    "Write : WRITE '(' STRING ')'"
    global output
    output += ("pushs " + p[3] + "\n")
    output += ("writes \n")


def p_Write_Var(p):
    "Write : WRITE '(' VAR ')'"
    global output
    output += ("pushg " + str(p.parser.register.get(p[3])[0]) + "\n")
    output += ("stri\n")
    output += ("writes\n")


def p_Write_Array_Int(p):
    "Write : WRITE '(' VAR '[' INT ']' ')'"
    global output
    output += ("pushg " +
               str(p.parser.register.get(p[3])[0] + int(p[5])) + "\n")
```

```
    output += ("stri\n")
    output += ("writes\n")


def p_Write_Array_Var(p):
    "Write : WRITE '(' AtribArray ')'"
    global output
    output += ("loadn\n")
    output += ("stri\n")
    output += ("writes\n")

def p_Write_Matrix_Int(p):
    "Write : WRITE '(' VAR '[' INT ']' '[' INT ']' ')'"
    var = p.parser.register.get(p[1])
    global output
    output += ("pushg " + str(var[0] + var[2] * int(p[3]) + (int(p[6]))) + "\n")
    output += ("stri\n")
    output += ("writes\n")


def p_Write_Matrix_Var(p):
    "Write : WRITE '(' AtribMatrix ')'"
    global output
    output += ("loadn\n")
    output += ("stri\n")
    output += ("writes\n")

# ------------------------------------------------------------------------------
# IfStatement ------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_IfStatement(p):
    "IfStatement : If '{' Body '}'"
    global output
    output += ("endif" + str(condStack.pop()) + ":" + "\n")

# ------------------------------------------------------------------------------
# If ---------------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_If(p):
    "If : IFID '(' Cond ')'"
    global condCounter, condStack, output
    condStack.append(condCounter)
    output += ("jz endif" + str(condCounter) + "\n")
    condCounter += 1

# ------------------------------------------------------------------------------
#  Body ------------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_Body_Single(p):
    "Body : Command"


def p_Body_List(p):
```

```
    "Body : Body Command"

# ------------------------------------------------------------------------------
# ForStatement -----------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_ForStatement(p):
    "ForStatement : For '{' Body '}'"
    global output
    output += ("jump repeat" + str(condStack[(len(condStack)-1)]) + "\n")
    output += ("repeatend" + str(condStack.pop()) + ":\n")

# ------------------------------------------------------------------------------
# For --------------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_For(p):
    "For : FORID '(' Atrib InitLabel ForCond ')'"

# ------------------------------------------------------------------------------
# InitLabel --------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_InitLabel(p):
    "InitLabel : ';'"
    global condCounter, condStack, output
    condStack.append(condCounter)
    output += ("repeat" + str(condCounter) + ":\n")
    condCounter += 1

# ------------------------------------------------------------------------------
# ForCond ----------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_ForCond(p):
    "ForCond : Cond"
    global output
    output += ("jz repeatend" + str(condStack[(len(condStack)-1)]) + "\n")

# ------------------------------------------------------------------------------
# Cond -------------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_Cond_Equals(p):
    "Cond : Factor '=' '=' Factor"
    global output
    output += ("equal\n")


def p_Cond_NotEquals(p):
    "Cond : Factor '!' '=' Factor"
    global output
    output += ("equal\nnot\n")


def p_Cond_Greater(p):
```

```
    "Cond : Factor '>' Factor"
    global output
    output += ("sup\n")


def p_Cond_GreaterEquals(p):
    "Cond : Factor '>' '=' Factor"
    global output
    output += ("supeq\n")


def p_Cond_MinorEquals(p):
    "Cond : Factor '<' '=' Factor"
    global output
    output += ("infeq\n")


def p_Cond_Minor(p):
    "Cond : Factor '<' Factor "
    global output
    output += ("inf\n")

# ------------------------------------------------------------------------------
# Expr -------------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_Expr_add(p):
    "Expr : Expr '+' Term"
    global output
    if (error != 1):
        output += ("add\n")


def p_Expr_sub(p):
    "Expr : Expr '-' Term"
    global output
    if (error != 1):
        output += ("sub\n")


def p_Expr_Term(p):
    "Expr : Term"

# ------------------------------------------------------------------------------
# Term -------------------------------------------------------------------------
# ------------------------------------------------------------------------------

def p_Term_mull(p):
    "Term : Term '*' Factor"
    global output
    if (error != 1):
        output += ("mul\n")


def p_Term_div(p):
    "Term : Term '/' Factor"
```

```
    global output
    if (error != 1):
        output += ("div\n")


def p_Term_mod(p):
    "Term : Term '%' Factor"
    global output
    if (error != 1):
        output += ("mod\n")


def p_Term_Factor(p):
    "Term : Factor"

# ------------------------------------------------------------------------
# Factor -----------------------------------------------------------------
# ------------------------------------------------------------------------

def p_Factor_INT(p):
    "Factor : INT"
    global output
    if (error != 1):
        output += ("pushi "+p[1]+"\n")


def p_Factor_VAR(p):
    "Factor : VAR"
    global output, errorCounter, error
    if(p.parser.register.get(p[1]) == None):
        error = 1
        output += ("err \"variable '" + p[1] + "' is not defined\"\n")
        errorCounter += 1
    elif (p.parser.register.get(p[1])[1] > 1):
        error = 1
        output += ("err \"Unsupported operand type(s)\"\n")
        errorCounter += 1
    else:
        output += ("pushg " + str(p.parser.register.get(p[1])[0]) + "\n")


def p_Factor_VAR_Int(p):
    "Factor : VAR '[' INT ']'"
    global output, errorCounter, error
    if(p.parser.register.get(p[1]) == None):
        error = 1
        output += ("err \"variable '" + p[1] + "' is not defined\"\n")
        errorCounter += 1
    elif(p.parser.register.get(p[1])[1] == 1):
        error = 1
        output += ("err \"'int' object is not subscriptable\"\n")
        errorCounter += 1
    else:
        output += ("pushg " +
                   str(p.parser.register.get(p[1])[0] + int(p[3])) + "\n")
```

```
def p_Factor_VAR_Int_Int(p):
    "Factor : VAR '[' INT ']' '[' INT ']'"
    global output, errorCounter, error
    var = p.parser.register.get(p[1])
    if(p.parser.register.get(p[1]) == None):
        error = 1
        output += ("err \"variable '" + p[1] + "' is not defined\"\n")
        errorCounter += 1
    elif(p.parser.register.get(p[1])[1] == 1):
        error = 1
        output += ("err \"'int' object is not subscriptable\"\n")
        errorCounter += 1
    else:
        output += ("pushg " + str(var[0] + var[2] * int(p[3]) + (int(p[6]))) + "\n")


def p_Factor_group(p):
    "Factor : '(' Expr ')'"


def p_error(p):
    stack_state_str = ' '.join([symbol.type for symbol in parser.symstack][1:])

    print('Syntax error in input! Parser State:{} {} . {}'
          .format(parser.state,
                  stack_state_str,
                  p))

# ------------------------------------------------------------------------------
# ------------------------------------------------------------------------------


parser = yacc.yacc()

parser.register = {}

parser.counter = 0
inFile = open("test.txt")

codeBlock = False
aux = ""
chavetas = 0

for linha in inFile:
    error = 0

    if ("{") in linha:
        codeBlock = True
        chavetas += 1

    if ("}") in linha:
        codeBlock = False
        chavetas -= 1

    aux += linha
```

```
    if chavetas == 0:
        result = parser.parse(aux)
        aux = ""

inFile.close()

print(parser.register)
outFile = open("output.txt", "w", encoding="ascii")

if(errorCounter > 0):
    errors = re.findall(r'err \"[^\"]+\"[ ]*\n', output)
    output = ''.join(errors)

outFile.write(output)
outFile.close()
```

8.3 Gramática

```
Prog -> InitBlock
Prog -> InstrBlock

InitBlock -> Init
           | InitBlock Init

Init -> INTID VAR
      | INTID '[' INT ']' VAR
      | INTID '[' INT ']' '[' INT ']' VAR

InstrBlock -> Command
           | InstrBlock Command

Command -> Atrib
         | Read
         | Write
         | IfStatement
         | ForStatement

Atrib -> VAR '=' Expr
       | VAR '=' Read
       | VAR '[' INT ']' '=' Expr
       | AtribArray '=' Expr
       | VAR '[' INT ']' '=' Read
       | AtribArray '=' Read
       | VAR '[' INT ']' '[' INT ']' '=' Expr
       | VAR '[' INT ']' '[' INT ']' '=' Read
       | AtribMatrix '=' Expr
       | AtribMatrix '=' Read

AtribArray -> VAR '[' VAR ']'

AtribMatrix-> VAR '[' VAR ']' '[' VAR ']'

Read -> READ '(' ')'

Write -> WRITE '(' STRING ')'
       | WRITE '(' VAR ')'
       | WRITE '(' VAR '[' INT ']' ')'
       | WRITE '(' AtribArray ')'
       | WRITE '(' VAR '[' INT ']' '[' INT ']' ')'
       | WRITE '(' AtribMatrix ')'

IfStatement -> If '{' Body '}'

If -> IFID '(' Cond ')'

Body -> Command
      | Body Command

ForStatement -> For '{' Body '}'

For -> FORID '(' Atrib InitLabel ForCond ')'
```

```
InitLabel -> ';'

ForCond -> Cond

Cond -> Factor '=' '=' Factor
      | Factor '!' '=' Factor
      | Factor '<' '=' Factor
      | Factor '>' '=' Factor
      | Factor '>' Factor
      | Factor '<' Factor


Expr -> Expr '+' Termo
      | Expr '-' Termo
      | Termo

Termo -> Termo '*' Factor
        | Termo '/' Factor
        | Termo '%' Factor
        | Factor

Factor -> '(' Expr ')'
       | INT
       | VAR
       | VAR '[' INT ']'
       | VAR '[' INT ']' '[' INT ']'
```