Processamento de Linguagens (3º ano de Curso)

Trabalho Prático 2

Relatório de desenvolvimento

Grupo 18

João Correia (A84414)

Marco Pereira (A89556)

28 de junho de 2021

Resumo

O presente relatório tem como objetivo descrever o processo de desenvolvimento da ferramenta *JomaCompiler*, um compilador da linguagem *Joma* que irá receber como input um ficheiro da linguagem Joma e irá produzir um ficheiro com a extensão vm com instruções em pseudo-código, Assembly da Máquina Virtual VM.

O trabalho desenvolvido concretizou-se numa ferramenta capaz de converter um ficheiro de extensão **.jm**, interpretando várias implementações de ciclos, declarações de funções, arrays, conversões.

Embora o escopo da ferramenta seja amplamente maior do que o originalmente proposto, o que resulta num sistema de complexidade considerável, este retém, no entanto, uma elevada eficácia de processamento.

Conteúdo

Capítulo 1

Introdução

1.1 JomaCompiler

Área: Processamento de Linguagens

O seguinte projeto encontra-se inserido no contexto da unidade curricular de Processamentos de Linguagens, presente no 3º ano letivo do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

Este constitui o segundo trabalho prático presente na componente prática da avaliação da unidade curricular, cujo objetivos avaliativos se centram na capacidade de desenvolvimento de *Expressões Regulares (ER)* de descrição de padrões textuais e no desenvolvimento de sistemas capazes de transformar texto numa lógica *condição-ação*. Este tem, como suporte, a linguagem de programação Python, utilizando o módulo *re* [1], que disponibiliza diversas ferramentas de reconhecimento e transformação textual através do uso de expressões regulares. Além disso irá utilizar o ply do Python para obter acesso ao módulo yacc e lex.

O tema a abordar será o desenvolvimento de um compilador capaz de converter ficheiros de uma linguagem de programação criada pelos alunos, denominada Joma, num fiheiro com instruções de pseudo-código, Assembly da Máquina Virtual VM fornecida pela UC. O ficheiro fornecido ao compilador poderá conter vários tipos de instruções e funcionalidades como ciclos, condições, funções, arrays, declarações locais, casts, entre outros. Além disso o ficheiro pode conter variáveis do tipo float e inteiro, desta forma, a linguagem criada pelo grupo contém um elevado número de fucnionalidades que serão descritas em secções futuras do presente relatório.

Este relatório terá como objetivo a descrição detalhada dos requisitos do projeto, assim como o de detalhar o processo de concepção e implementação do algoritmo responsável pela interpretação do ficheiro fornecido sublinhando as suas funcionalidades, tanto as especificadas como requeridas no enunciado do projeto, como as funcionalidades adicionais reconhecidas como úteis por parte do grupo responsável pelo desenvolvimento do compilador.

O compilador desenvolvido revela-se eficaz na tarefa proposta, contendo as funcionalidades necessárias para que a linguagem que interpreta seja útil e possa ser utilizada como forma de criar programas simples com funcionalidades básicas.

Estrutura do Relatório

O presente relatório encontra-se dividido em 5 secções

A primeira secção, afigura-se como a introdução ao trabalho, secção onde a atual descrição da estrutura do relatório se encontra presente.

No segundo capítulo, 2, o projeto proposto é analisado, sendo levantado um conjunto de requisitos, tanto mencionadas no enunciado deste como também requisitos adicionais inferidos como úteis pela equipa responsável pelo projeto.

O terceiro capítulo, 3, irá descrever a forma como foi desenvolvida a ferramenta, apresentando as variáveis de compile time, os vários símbolos utilizados, terminais e não terminais, bem como as regras gramaticais.

No capítulo 4, o processamento descrito no capítulo anterior é implementado sob a forma de um programa desenvolvido na linguagem de programação Python. Aqui são apresentados os detalhes técnicos do funcionamento da ferramenta bem como apresentados alguns testes e problemas encontrados na máquina virtual durante os mesmos.

Por fim, o quinto capítulo, 5, conclui o relatório, sintetizando a informação disponibilizada ao longo deste.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

O sistema a desenvolver deverá ser um compilador de ficheiros gravados em formato *Joma* (uma linguagem criada pelo grupo com a extensão *jm*) para ficheiros em formato *Máquina Virtual VM* (um conjunto de instruções em pseudo-código Assembly funcional na máquina virtual cedida pelos docentes da Unidade Curricular), através da utilização do módulo re e ply do Python para reconhecer vários tipos de funcionalidades na linguagem, sendo exemplo disso, a implementação de 3 tipos diferentes de ciclos, funções com declarações locais, arrays uni e bidimensionais, condições com expressões condicionais encadeadas, diferentes tipos de variáveis e casts feitos automaticamente entre elas.

2.2 Especificação do Requisitos

Do enunciado referido na secção anterior é possível levantar um conjunto de requisitos que o projeto deverá cumprir. O grupo, no entanto, julgou-os insuficientes para o desenvolvimento de uma ferramenta completa, pelo que foi levantado um conjunto de requisitos adicionais que tem o objetivo de complementar o funcionamento da ferramenta.

2.2.1 Requisitos base

Os requisitos base são os que é possível extrair do enunciado presente na secção 2.1. Estes são, de seguida, enunciados.

1. Declarar variáveis atómicas do tipo inteiro, com os quais se podem realizar as habituais operaçõoes aritméticas, relacionais e lógicas;
2. Efetuar instruções algorítmicas básicas como a atribuição do valor de expressões numéricas a variáveis;
3. Ler do standard input e escrever no standard output;
4. Efetuar instruções condicionais para controlo do fluxo de execução;
5. Efetuar instruções cíclicas para controlo do fluxo de execução, permitindo o seu aninhamento;
6. Definir e invocar subprogramas sem parâmetros mas que possam retornar um resultado do tipo inteiro.

2.2.2 Requisitos adicionais

Embora os requisitos acima mencionados constituam uma base sólida para um conversor, um conjunto de funcionalidades necessárias para que este seja uma ferramenta completa encontram-se em falta. Como tal, o grupo decidiu inclui-los no levantamento de requisitos. Estes são:

1. Definir a existência de 3 tipos de variáveis, sendo elas, int, float e string;
2. Definir a ocorrência de casts automáticos em caso de atribuições indevidas;
3. Definir a exitência de 3 tipos diferentes de ciclos, sendo eles os 3 tipos aconselhados pelos docentes da Unidade Curricular;
4. Permitir a declaração de variáveis locais dentro de subprogramas;
5. Permitir o encademaneto de condições com a utilização de operações de interseção e/ou união;
6. 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)(desenvolvendo assim as 2 funcionalidades adicionais).

Um exemplo simples com alguns dos requisitos referenciados em cima encontra-se em C, onde podemos também ver um exemplo mais extenso que será referido na fase de testes.

Capítulo 3

Concepção/desenho da Resolução

Devido à existência de funcionalidades como ciclos condições e funções decidimos optar pelo processamento do ficheiro na totalidade pois existem algumas dependências entre instruções, como seria o caso de instruções que fazem parte de um ciclo, isto é, serão repetidas várias vezes, e instruções que se encontram fora do ciclo, esta separação foi feita uma grande parte das vezes com recorrência a chavetas({,}).
Como tal, iremos abordar primeiramente os atributos que achamos necessários ter no parser e a sua funcionalidade, iremos depois passar para a apresentação das regras da gramática e das consequências da mesma no resultado da compilação.

3.1 Descrição do Parser

Como é habitual neste tipo de exercícios, em conjunto com o *yacc* iremos utilizar um *parser*, este parser vai ter como objetivo guardar a informação ao longo da compilação de modo a ser possível adquirir os valores dos registos de cada variável, entre outros.
Como tal, os atributos colocados no parser são os seguintes:

success

Success será um booleano com valor verdadeiro caso a compilação termine de forma bem sucedida, e falso caso encontre algum erro que não estava à espera.

compiled

Compiled será uma string que irá conter toda a informação compilada até ao momento que será escrita posteriormente para o ficheiro caso a compilação seja bem sucedida.

current_type

Current_type será uma string que irá sinalizar o tipo com que se está a lidar atualmente na compilação, o seu uso será maioritariamente para a implementação dos casts feitos automaticamente, terá os seguintes valores:

- "": o tipo atual é um inteiro;
- "f": o tipo atual é um float;

- ”s”: o tipo atual é uma string;
- None : não existe tipo atual.

current_func

Current_func é também uma string e irá conter o nome da função a compilar atualmente, desta forma será possível controlar as variáveis locais a que a função tem acesso e também será possível produzir mensagens de erro com mais informação. Toma o valor ”0”quando a função atual é nenhuma, isto é, global.

func_var_counter

Func_var_counter será um dicionário com chave string representante do nome da função, e valor inteiro, este inteiro será responsável por guardar a linha onde a variável da função foi declarada, inicia-se a 0 e será iterado a cada variável inserida na função correspondente.

func_vars

Func_vars será um dicionário com chave string representante do nome da função, e valor dicionário que mapea as variáveis declaradas na função pelo nome para o inteiro correspondente à sua localização na stack em relação ao *framepointer* (ou *globalpointer* caso seja uma variável global).

func_var_types

Func_var_types funciona da mesma forma do que o anterior mas tem o intuito de guardar o tipo da variável que mapea, esse tipo está guardado num formato string seguindo o guia acima apresentado em current_type.

func_array_info

Func_array_info será responsável por guardar para cada função um dicionário que contém um mapeamento de nome de variável para um array de duas posições onde será colocado o número de linhas e colunas do array declarado. O número de colunas será 1 para arrays que não sejam bidimensionais.

3.2 Apresentação da gramática

Terminada a explicação dos vários atributos do parser vamos agora apresentar a grámatica, começando pelos símbolos terminais, seguindo para os símbolos não terminais e terminando com as várias regras da grámatica por nós definida, apresentando o raciocínio que levou a cada uma.

3.2.1 Símbolos Terminais

Os símbolos terminais da nossa linguagem, bem como as expressões regulares dos mesmos estão presentes em A

Como podemos ver, optamos pela utilização de $<$ e $>$ para marcar cada um dos identificadores como *if*, *while*, *for*. O grupo achou que desta forma a linguagem se mantinha fácil de interpretar e manteve a congruência da mesma ao longo de todos os identificadores. Além disso conseguimos perceber que existem identificadores para os varios tipos de dados que seguem as mesmas especificações encontradas na documentação da máquina virtual.

3.2.2 Símbolos não Terminais

Os símbolos não terminais da nossa linguagem aparecem abaixo listados e serão descritos de forma mais aprofundada em secções seguintes em conjunto com as regras gramaticais da linguagem.

Array	Condicao	EndFunc	Repeat
ArrayCol	ContArray	Escrita	Ret
Atribuicao	ContContinuacaoVar	Expr	Start
AtribuicaoInicial	ContinuacaoVar	ExprCond	Term
Ciclo	DecFunc	Factor	Type
CicloFor	Declaracao	Funcao	Until
CicloRepeat	Declaracoes	Inicio	Var
CicloWhile	Do	Instrucao	Variaveis
Codigo	Else	Leitura	While
Cond	EndCond	ListInstrucao	

3.2.3 Regras gramaticais

Após terem sido apresentados os vários símbolos terminais e não terminais podemos agora apresentar as várias regras gramaticais, estas encontram-se abaixo:

```
Rule 0 S' -> Inicio
Rule 1 Inicio -> Variaveis Start Codigo
Rule 2 Start -> <empty>
Rule 3 Variaveis -> { Declaracoes }
Rule 4 Declaracoes -> Declaracoes Declaracao
Rule 5 Declaracoes -> <empty>
Rule 6 Declaracao -> Type ID ;
Rule 7 Declaracao -> Type ID = Expr ;
Rule 8 Declaracao -> Type Array ID ;
Rule 9 Type -> ID
Rule 10 Array -> [ INT ] ContArray
Rule 11 ContArray -> [ INT ]
Rule 12 ContArray -> <empty>
Rule 13 Expr -> Expr + Term
Rule 14 Expr -> Expr - Term
Rule 15 Expr -> Term
Rule 16 Term -> Term * Factor
Rule 17 Term -> Term / Factor
Rule 18 Term -> Term % Factor
Rule 19 Term -> Factor
Rule 20 Factor -> INT
Rule 21 Factor -> - INT
Rule 22 Factor -> FLOAT
Rule 23 Factor -> - FLOAT
Rule 24 Factor -> STRING
Rule 25 Factor -> CALL ID
Rule 26 Factor -> ID
```

Rule 27 Factor -> ID [INT] ArrayCol
Rule 28 ArrayCol -> [INT]
Rule 29 ArrayCol -> <empty>
Rule 30 Codigo -> Codigo Funcao
Rule 31 Codigo -> Codigo Instrucao
Rule 32 Codigo -> <empty>
Rule 33 Funcao -> DecFunc { Variaveis ListInstrucao EndFunc
Rule 34 DecFunc -> FUNC ID
Rule 35 EndFunc -> }
Rule 36 Ret -> RET Expr ;
Rule 37 ListInstrucao -> ListInstrucao Instrucao
Rule 38 ListInstrucao -> <empty>
Rule 39 Instrucao -> Atribuicao
Rule 40 Instrucao -> Leitura
Rule 41 Instrucao -> Escrita
Rule 42 Instrucao -> Condicao
Rule 43 Instrucao -> Ciclo
Rule 44 Instrucao -> Ret
Rule 45 Atribuicao -> Var = Expr ;
Rule 46 Var -> ID ContinuacaoVar
Rule 47 ContinuacaoVar -> [INT] ContContinuacaoVar
Rule 48 ContinuacaoVar -> <empty>
Rule 49 ContContinuacaoVar -> [INT]
Rule 50 ContContinuacaoVar -> <empty>
Rule 51 Leitura -> READ Var ;
Rule 52 Escrita -> WRITE Expr ;
Rule 53 Condicao -> IF ExprCond Do { ListInstrucao } ;
Rule 54 Condicao -> IF ExprCond Do { ListInstrucao } Else { ListInstrucao } ;
Rule 55 Do -> DO
Rule 56 Else -> ELSE
Rule 57 ExprCond -> ExprCond & Cond
Rule 58 ExprCond -> ExprCond | Cond
Rule 59 ExprCond -> Cond
Rule 60 Cond -> Expr EQUAL Expr
Rule 61 Cond -> Expr NOTEQUAL Expr
Rule 62 Cond -> Expr BIGGER Expr
Rule 63 Cond -> Expr SMALLER Expr
Rule 64 Cond -> Expr BIGGEREQUAL Expr
Rule 65 Cond -> Expr SMALLEREQUAL Expr
Rule 66 Ciclo -> CicloWhile
Rule 67 Ciclo -> CicloRepeat
Rule 68 Ciclo -> CicloFor
Rule 69 CicloWhile -> While ExprCond Do { ListInstrucao }
Rule 70 While -> WHILE
Rule 71 CicloRepeat -> Repeat ListInstrucao Until ExprCond
Rule 72 Repeat -> REPEAT
Rule 73 Until -> UNTIL
Rule 74 CicloFor -> FOR (AtribuicaoInicial ExprCond EndCond Atribuicao) { ListInstrucao }
Rule 75 AtribuicaoInicial -> Atribuicao
Rule 76 EndCond -> ;

Como podemos ver acima, algumas das regras parecem irrelevantes, estas existem devido a ocorrências que queríamos que acontecessem primeiro, por exemplo, a existência do **Type** deve-se à necessidade de modificar o tipo atual (current_type) antes de proceder à continuação da declaração, desta forma precisamos de fazer essa alteração antes de executar o **Array** ou **Expr**, e a forma utilizada pelo grupo foi isolar o *ID* com a regra **Type**. Desta forma o grupo consegue também criar uma linguagem em BNF-Puro praticamente na totalidade com a exceção de alguns símbolos terminais literais.

Capítulo 4

Codificação e Testes

A ferramenta foi desenvolvida utilizando a linguagem de programação Python, recorrendo à biblioteca RE e PLY de forma a possibilitar a utilização de expressões regulares no processamento textual, assim como possibilitar a criação de analisadores léxicos.
O código encontra-se dividido em 4 blocos lógicos:

1. Lex;
2. Yacc;
3. Lógica de leitura e escrita do ficheiro .vm.
4. Testes Realizados e Resultados

4.1 Lex

Tendo em conta que os símbolos terminais foram já apresentados iremos agora explicar as intenções dos mais relevantes e a expressão regular correspondente.
As expressões regulares mais importantes e nas quais se baseia grande parte do trabalho são as dos tipos de dados que a nossa linguagem de programação suporta, sendo assim temos as 3 seguintes expressões regulares:

t_FLOAT = r'\d+\.\d+'
t_INT = r'\d+'
t_STRING = r'\"([^¨]|\")*\"'

Após estas vemos que todos os outros símbolos terminais são auto-explicativas estando apenas entre '<' e '>'.
Como tal podemos dar como explicados os símbolos terminais, mais informação sobre eles pode ser vista em A.

4.2 Yacc

O algoritmo de processamento do Yacc será descrito nesta secção. A sua implementação encontra-se disponível no anexo B.
Tendo sido descritos na secção 3.1, iremos, agora, apenas apresentar a metodologia utilizada na resolução do exercício proposto e em cada uma das suas fases.

4.2.1 Declaração de variáveis

Devido à restrição de que as variáveis devem ser todas declaradas no início do programa sabemos que estas vão estar todas localizadas no fundo da stack, como tal podemos atribuir a posição 0 da stack à primeira váriavel que declararmos, a posição 1 à segunda e assim sucessivamente, a declaração de arrays será descrita mais à frente. No caso de declarações de variáveis locais devemos utilizar a instrução pushg X para aceder ao seu valor sendo X a posição onde esta se encontra em relação ao *global pointer*. No caso de variáveis locais devemos usar a instrução pushl X para aceder ao seu valor sendo X a posição onde esta se encontra em relação ao *frame pointer*.

4.2.2 Atribuição de valores a variáveis

A atribuição de valores a variáveis será feita tendo em conta o reultado proveniente do símbolo não terminal **Expr**. A formulação por trás desse símbolo não terminal é a mesma utilizada nas aulas tendo apenas algumas adições como a operação que calcula o resto da divisão inteira, uma funcionalidade de conversão de tipos, a utilização da instrução concat para "somas"entre strings, etc. Após todas as instruções que determinam o resultado de **Expr** estarem escritas iremos acrescentar as instruções que determinam a conversão de tipos para que o resultado da **Expr** esteja de acordo com o tipo da variável à qual esta vai ser atribuída, esta funcionalidade utiliza o tipo da variável e o tipo da **Expr** para decidir a conversão a ser feita, depois disto iremos acrescentar a instrução storeY X, em que Y é 'g' ou 'l' consoante o ambiente onde a variável à qual vai ser atribuído o valor foi declarada e X a localização da stack onde a variável se encontra.

4.2.3 Leitura do standard input e escrita para o standard output

Leitura

Este requisito, embora simples de implementar, tem uma nuance que passa por duplicar o endereço da string lida antes de proceder à conversão de tipos, esta duplicação tem como objetivo permitir a libertação de memória após a atribuição dos valores lidos. As instruções utilizadas para a implementaçao desta funcionalidade foram read, atoi/atof, dup, free e storeg/storel.

Escrita

O requisito de escrita é bastante simples de implementar sendo apenas necessário a utilização das instruções pushg/pushl e writei/writef/writes. Para esta funcionalidade temos apenas que dar push da variável ou valor que queremos imprimir no ecrã seguido da instrução de escrita que irá consumir esse valor e imprimir o mesmo no ecrã.

4.2.4 Instruções condicionais

Esta funcionalidade consiste na implementação de condições e de instruções condicionais, estas apresentam-se na forma de um if-do-else. Como tal serão utilizadas labels para poder proceder aos saltos condicionais, abaixo temos uma pequena demostração de como isto será implementado em pseudo-código.

```
Condição
   jz label_else
      Código que verifica a condição
   jump label_fim
label_else :
      Código que não verifica a condição
label_fim :
```

Como tal temos que criar 2 labels para a implementação desta funcionalidade, uma para o salto para o código do else, outra para o salto após correr o código if, o mesmo se aplica caso tenhamos apenas a formulação if-do, no entanto este não irá necessitar de uma instrução de jump após a realização do bloco de instruções que verificam a condição proposta. A implementação de condições a verificar passa pelo uso dos operadores '&' e '|' como 'e' e 'ou', respetivamente. Quando estamos perante um '&' iremos executar a instrução de multiplicação dos valores obtidos entre as 2 expressões separadas por este, obtendo 0 caso algum deles seja 0(falso) e apenas obtendo 1 quando ambos tem o valor 1(verdadeiro). Quando estamos perante um '|' iremos executar a instrução de adição dos valores obtidos, desta forma, quando os valores das expressões que operador separa são 0 obtemos 0 e quando o valor que separa as instruções é 1 obtemos sempre um valor diferente de 0.

A implementação de labels foi feita através da utilização de uma estrutura de dados entitulada stack, desta forma conseguimos carregar para a stack cada label e descarregando cada endereço de forma a que a utilização de condições aninhadas seja possível.

4.2.5 Instruções cíclicas

A funcionalidade que vamos agora descrever estará dividida em 3 secções, sendo que foram implementados 3 tipos de ciclos.

Ciclo While-Do

A funcionalidade do ciclo while-do segue o principio de que enquanto uma definida expressão condicional for verificada irá ocorrer um bloco de código, geralmente este bloco de código contém uma instrução que direta ou indiretamente altera o resultado da condição de forma a que este ciclo tenha fim. Este ciclo segue a implementação apresentada abaixo em pseudo-código.

```
label_inicio_ciclo :
Condição
   jz label_fim_ciclo
      Código que verifica a condição
   jump label_inicio_ciclo
label_fim_ciclo :
```

Desta forma, com a ajuda da regra gramatical que define o ciclo e com a stack onde as labels vão sendo empilhadas e removidas podemos proceder à implementação da funcionalidade com facilidade, começando por escrever uma label e empilhar, escrever a instrução jz com uma nova label e empilhar esta também, adicionar o código dentro do ciclo while como já foi visto anteriormente, sendo que este será delimitado por '' e '' torna-se simples saber as limitações onde se inicia e termina, e após isso temos apenas que fazer dois pop's da stack de labels sendo o primeiro a **label_fim_ciclo** e a segunda a **label_inicio_ciclo**, pelo exemplo acima vemos que temos que inserir duas frases com a ordem dos pop's invertida.

Ciclo Repeat-Until

A funcionalidade do ciclo repeat-until segue o principio de que até que uma definida expressão condicional for verificada irá ocorrer um bloco de código, geralmente este bloco de código contém uma instrução que direta ou indiretamente altera o resultado da condição de forma a que este ciclo tenha fim. A implementação deste ciclo pode ser descrita pelo excerto em pseudo-código abaixo.

```
jump label_condicao
label_inicio_ciclo :
    Código que não verifica a condição
label_condicao :
Condição
  jz label_inicio_ciclo
```

Com a ajuda das regras gramaticais produzidas para o efeito e com a stack de labels já referida anteriormente podemos seguir uma metodologia proxima da que seguimos no ciclo while-do empilhando a label de condição após a escrita do jump para a mesma, empilhando a label do código no ciclo escrevendo também a mesma, após isto escrevemos apenas o código que se encontra entre repeat e until de formas já antes mencionadas tendo apenas a necessidade de fazer dois pop's seguidos da stack para poder imprimir a label de condição que seria o segundo pop seguido do código que define a expressão condicional terminando por fim com a escrita da última linha utilizando a label de inicio de ciclo sendo esta resultante do primeiro pop feito à stack.

Ciclo For-Do

A funcionalidade do ciclo for-do segue a estruturação que temos na linguagem de programação C, desta forma teremos inicialmente uma condição inicial de atribuição a uma variável, seguido de uma expressão condicional, seguido de uma atribuição que será feita no fim de cada iteração do ciclo. A formulação deste ciclo pode se descrita pelo excerto abaixo apresentado.

```
    Atribuição incial
label_condição :
Condição
  jz label_fim_ciclo
    Código dentro de ciclo
    Código de atribuição após cada iteração
jump label_condicao
label_fim_ciclo :
```

Através da estrutura apresentada acima vemos que são apenas usadas 2 labels, para tal precisamos de proceder à escrita da atribuição no fim de cada ciclo logo após ao código que iria correr dentro do ciclo, tirando isto a metodologia seguida é a mesma apresentada nos ciclos anteriores, tirando partido de sucessivos push's e pop's de modo a permitir o aninhamento de vários ciclos.

4.2.6 Declaração e Manuseamento de Arrays

Declaração

A declaração deste tipo de variáveis segue uma regra gramatical de forma a obter o número de linhas e colunas, caso existam, após isso a multiplicação do número de linhas pelo número de colunas irá fornecer a quantidade de espaços que o array irá necessitar, desta forma temos apenas que escrever uma declaração

inicial de cada um dos constituintes do array e devemos tomar em atenção que o contador de variáveis deverá ser incrementado no número total de constituintes do array e não apenas em 1 como temos vindo a fazer com as outras variáveis.

Manuseamento

A implementação desta funcionalidade foi bastante simples tendo em conta que foi apenas necessário perceber o constituinte do array ao qual estávamos a aceder, para tal acedemos primariamente aos dados do array em questão para saber o número de linhas e colunas, estes dados são guardados na fase de declaração, após isso fazemos algum controlo de erros no caso de acesso a índices de memória não reservados para o array e por fim fazemos uma simples conta através da qual obtemos a posição em memória à qual devemos aceder, esta expressão é:

$$posicãoemmemória = posicãodoprimeiroelemento + linha\_acedida * númerodecolunas + coluna\_acedida$$

Em caso de acesso ao valor devemos utilizar a instrução de pushg/pushl e em caso de alteração do valor devemos utilizar a instrução de storeg/storel.

4.2.7 Definição e invocação de subprogramas(funções)

Definição

A definição de um subprograma é feita tendo em conta uma mistura de especificações Python com C, desta forma a identificação de uma função é feita com o identificador <func> seguido do nome da função e dentro de chavetas todo o código que diz respeito a esta. As instruções de retorno devem ser iniciadas pelo identificador <ret> e é necessário existir pelo menos uma, podendo no entanto existir mais do que uma para permitir retornos a meio de ciclos de modo a melhorar a performance de qualquer código gerado. Desta forma vemos abaixo um excerto em pseudo-código de como deverá ficar.

```
  jump label_depois_funcao
  label_nome_funcao
Código de declaração de variáveis locais
Código da função
  label_depois_funcao :
```

O código da função poderá conter linhas de return, estas são expressas por storel -1 seguido de return.

Todas as instruções descritas são simples de entender sendo apenas de salientar que as variáveis declaradas localmente devem ser manuseadas utilizando pushl e storel ao contrário das que são utilizadas mas foram declaradas globalmente.

Invocação

Após a declaração de uma função temos apenas que proceder à chamada desta, para tal devemos seguir a estrutura descrita abaixo, de notar que os nomes utilizados seguem a terminologia acima apresentada.

```
  push valor onde será guardado o resultado
  pusha label_nome_funcao
  call
Linhas de utilização do valor, podem ser write ou atribuições
```

4.3 Lógica de leitura e escrita do ficheiro .vm.

Uma das variáveis apresentadas tem o nome de compiled e como foi referido na sua explicação esta serve para guardar o conteúdo daquilo que desejamos escrever para o ficheiro .vm, a escrita é apenas feita no fim pois apenas sabemos se a compilação foi bem sucedida nessa altura. Neste sentido evitamos a destruição de compilações anteriores, com compilações não funcionais.

A chamada ao programa deve ser feita dando no mínimo 1 argumento que identifica o ficheiro a compilar, com possibilidade de ser dados 2 argumentos em que o segundo identifica o ficheiro para o qual será escrito o resultado da compilação, em caso de omissão será escrito para o ficheiro a.vm.

4.4 Testes realizados e Resultados

De modo a testar o programa foi criado um ficheiro escrito na linguagem por nós criada, este ficheiro contém os vários testes pedidos, cada um definido no seu subprograma, existindo posteriormente um menu para o utilizador escolher o programa que deseja testar.
O código está apresentado em C, onde o seu output em pseudo-código assembly está também presente.

4.4.1 Problemas encontrados com a Virtual Machine

Ao longo da realização do trabalho foram encontrados alguns problemas com a máquina virtual, alguns deles são apenas visuais e em nada afetam o decorrer do programa, no entanto existe um problema que após ter sido testado intensivamente se revelou estar na máquina virtual.

Problemas Visuais

Quando é feito um cast de float para int a máquina virtual(versão gráfica) apresenta esse resultado como float e com o valor 0.00000, no entanto quando este valor é usado para uma adição, por exemplo, obtemos o resultado suposto.
Na máquina virtual, na heap onde são guardadas as strings não são eliminadas strings antigas mesmo tendo sido dado free visto que elas continuam visíveis na heap, no entanto o contador dessa heap volta a 0, isto é apenas um bug visual que apenas confunde a utilização do free pois na versão gráfica a heap poderia/deveria ficar vazia de modo a ser mais percétivel a execução da instrução free.

Problema potenciador de erros

Quando é feito o push de uma string com apenas um caráter seguido de um push de uma string seguido de um concat, a string resultante do concat não possui a primeira string, sendo neste caso igual à segunda string. Este bug foi descoberto aquando da testagem do nosso programa pois é feito um push de um inteiro seguido de um cast deste para string, após um novo push de string é feito um concat e foi verificado que este concat não possuía o resultado da concatenação das duas strings mas sim apenas da segunda string empilhada, sendo que o inteiro usado tinha apenas 1 caráter.
Abaixo temos um pequeno excerto de código capaz de reproduzir este problema.

```
pushs "é impar"
pushs "1"
pushs "O valor "
concat
concat
writes
```

Com este código vm obtemos, como output no terminal: "O valor é impar"

```
pushs "é impar"
pushs "1.0"
pushs "O valor "
concat
concat
writes
```

Por outro lado com o código acima obtemos: "O valor 1. é impar"

Neste sentido o programa criado pelo grupo segue as normas inicialmente estabelecidas podendo não imprimir os resultados da forma mais correta devido a caráteres como o '\n' serem engolidos aquando do concat.

Capítulo 5

Conclusão

Conclui-se o presente relatório com uma apreciação positiva do trabalho realizado. A ferramenta de compilação desenvolvida +e capaz de interpretar programas escritos na linguagem Joma. A linguagem criada consegue reconhecer vários tipos de váriáveis, fazer casts automáticos entre eles, criar subprogramas e arrays de até 2 dimensões.

É aplicado, com sucesso, o conhecimento adquirido, até ao momento, na unidade curricular de Processamento de Linguagens [2], com especial ênfase na utilização de expressões regulares para deteção de padrões textuais e na utilização do PLY para produzir gramáticas de tradução coerentes e consistentes.

Como futuro trabalho sugere-se o suporte a mais funcionalidades de QoF, como a implementação do operador ++, +=,-=,*=,/=, utilização de variáves para aceder a posições de um array, entre outras.

Apêndice A

Código de processamento dos símbolos terminais

Listing A.1: Código de processamento dos símbolos terminais

```
1  import ply.lex as lex
2
3  tokens = ['ID', 'INT', 'FLOAT', 'STRING', 'READ', 'WRITE', 'EQUAL',  'NOTEQUAL', 'BIGGER', '
       BIGGEREQUAL',
4  'SMALLER', 'SMALLEREQUAL', 'RET', 'IF', 'DO', 'ELSE', 'WHILE', 'FUNC', 'CALL', 'REPEAT', 'UNTIL
       ', 'FOR']
5
6  literals = [';','=','+','-','*','/','%','{','}','&','|',':','(',')','[',']']
7
8  t_IF = r'<if>'
9
10 t_DO = r'<do>'
11
12 t_ELSE = r'<else>'
13
14 t_FUNC = r'<func>'
15
16 t_WHILE = r'<while>'
17
18 t_REPEAT = r'<repeat>'
19
20 t_UNTIL = r'<until>'
21
22 t_FOR = r'<for>'
23
24 t_CALL = r'<call>'
25
26 t_RET = r'<ret>'
27
28 t_READ = r'<read>'
29
30 t_WRITE = r'<write>'
31
32 t_EQUAL = r'=='
33
```

```
34 t_NOTEQUAL = r'!='
35
36 t_BIGGER = r'>'
37
38 t_BIGGEREQUAL = r'>='
39
40 t_SMALLER = r'<'
41
42 t_SMALLEREQUAL = r'<='
43
44 t_FLOAT = r'\d+\.\d+'
45
46 t_INT = r'\d+'
47
48 t_STRING = r'\"([^"]|\\\")*\"'
49
50 t_ID = r'[a-zA-Z]([a-zA-Z]|_|\d)*'
51
52 t_ignore = " \t\n"
53
54 def t_error(t):
55     print("Car ter ilegal: ", t.value[0])
56     t.lexer.skip(1)
57
58 lexer = lex.lex()
```

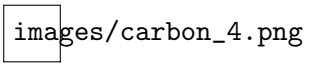

Figura A.1: Código ficheiro Lex

Apêndice B

Código de processamento presente no yacc

Listing B.1: Código de processamento presente no yacc

```
1  ' ' '
2  T:{ ID , INT , FLOAT , STRING , READ , WRITE , EQUAL , BIGGER , BIGGEREQUAL , SMALLER , SMALLEREQUAL , RET ,
       IF , DO , ELSE ,
3        WHILE , FUNC , CALL , REPEAT , UNTIL , FOR
             , ' ; ' , ' = ' , ' + ' , ' - ' , ' * ' , ' / ' , ' % ' , ' { ' , ' } ' , ' & ' , ' | ' , ' : ' , ' [ ' , ' ] ' }
4
5  N:{ Inicio , Start , Variaveis , Codigo , Declaracoes , Declaracao ,
6        Type , Array , ContArray , Expr , Term , Factor , Funcao ,
7        DecFunc , Ret , ListInstrucao , Instrucao , Atribuicao , Var ,
8        ContContinuacaoVar , Leitura , Escrita , Condicao , Ciclo ,
9        ExprCond , Do , Else , Cond , Ciclo , CicloWhile , While , Repeat , Until ,
10       CicloRepeat , CicloFor , AtribuicaoInicial }
11
12 S: Inicio
13
14 P:{
15       p1: Inicio => Start Variaveis Codigo
16       p2: Start =>
17       p3: Variaveis => ' { ' Declaracoes ' } '
18       p4: Declaracoes => Declaracoes Declaracao
19       p5:                |
20       p6: Declaracao => Type ID ' ; '
21       p7:               | ' = ' Expr ' ; '
22       p8:               | Array ID ' ; '
23       p9: Type => ID
24       p10: Array => ' [ ' INT ' ] ' ' [ ' INT ' ] '
25       p11:        |   ' [ ' INT ' ] '
26       p12: Expr => Expr ' + ' Term
27       p13:       |   Expr ' - ' Term
28       p14:       | Term
29       p15: Term => Term ' * ' Factor
30       p16:       |   Term ' / ' Factor
31       p17:       |   Term ' % ' Factor
32       p18:       |   Factor
33       p19: Factor => INT
```

```
    p20:          | '-' INT
    p21:          | FLOAT
    p22:          | '-' FLOAT
    p23:          | ID
    p24:          | ID '[' INT ']'
    p25:          | ID '[' INT ']' '[' INT ']'
    p26:          | CALL ID
    p27: Codigo => Funcao Codigo
    p28:          | Instrucao Codigo
    p29:          |
    p30: Funcao => DecFunc '{' Variaveis ListInstrucao Ret '}'
    p31: DecFunc => FUNC ID
    p32: Ret => RET Expr ';'
    p33: ListInstrucao => ListInstrucao Instrucao
    p34:                |
    p35: Instrucao => Atribuicao
    p36:            | Leitura
    p37:            | Escrita
    p38:            | Condicao
    p39:            | Ciclo
    p40: Atribuicao => Var '=' Expr ';'
    p41: Var => ID
    p42:      | ID '[' INT ']'
    p43:      | ID '[' INT ']' '[' INT ']'
    p44: Leitura => READ Var ';'
    p45: Escrita => WRITE Var ';'
    p46: Condicao => IF ExprCond Do '{' ListInstrucao '}' ';'
    p47:           | IF ExprCond Do '{' ListInstrucao '}' Else '{' ListInstrucao '}'
        ';'
    p48: Do => DO
    p49: Else => ELSE
    p50: ExprCond => ExprCond '&' Cond
    p51:           | ExprCond '|' Cond
    p52:           |
    p53: Cond => Expr EQUAL Expr
    p54:       | Expr BIGGER Expr
    p55:       | Expr BIGGEREQUAL Expr
    p56:       | Expr SMALLER Expr
    p57:       | Expr SMALLEREQUAL Expr
    p58: Ciclo => CicloWhile
    p59:        | CicloRepeat
    p60:        | CicloFor
    p61: CicloWhile => While ExprCond Do '{' ListInstrucao '}'
    p62: While => WHILE
    p63: CicloRepeat => Repeat '{' ListInstrucao '}' Until ExprCond
    p64: Repeat => REPEAT
    p65: Until => UNTIL
    p66: CicloFor => FOR '(' AtribuicaoInicial ExprCond EndCond Atribuicao ')' '{'
       ListInstrucao '}'
    p67: AtribuicaoInicial => Atribuicao
    p68: EndCond => ';'
}
'''
```

```
86 #Meter no expr a op   o de ser apenas '-' Expr que faria push 0 push p[2] e sub
87
88 import ply.yacc as yacc
89
90 from interpretador_lex import tokens,literals
91
92 def p_Inicio(p):
93     "Inicio : Variaveis Start Codigo"
94
95 def p_Start(p):
96     "Start : "
97     p.parser.compiled+="start\n"
98
99 def p_Variaveis(p):
100     "Variaveis : '{' Declaracoes '}'"
101
102 def p_Declaracoes(p):
103     "Declaracoes : Declaracoes Declaracao"
104
105 def p_Declaracoes_paragem(p):
106     "Declaracoes : "
107
108 def getAndIncFuncVarCounter(p,inc):
109     varNumAtual = p.parser.func_var_counter.get(p.parser.current_func)
110     p.parser.func_var_counter[p.parser.current_func] = varNumAtual + inc
111     return varNumAtual
112
113 def addVarToFunc(p,varname,inc):
114     p.parser.func_vars[p.parser.current_func][varname] = getAndIncFuncVarCounter(p,
          inc)
115     p.parser.func_var_types[p.parser.current_func][varname] = p.parser.current_type
116
117 def addVar(p,varName,inc):
118     linhaDec = p.parser.func_vars.get(p.parser.current_func).get(varName)
119     if linhaDec == None: #n o foi declarado na fun   o atual
120         linhaDec = p.parser.func_vars.get("0").get(varName) #pode ter sido declarada
              globalmente
121         if linhaDec is None: #n o foi declarada globalmente nem localmente logo pode
                ser adicionada
122             addVarToFunc(p,varName,inc)
123         else: #foi declarada globalmente e est  a ser redeclarada localmente
124             p.parser.success=False
125             error("Vari vel "+varName+" declarada localmente na fun   o "+p.parser.
                    current_func+" depois de ter sido declarada globalmente na "+str(
                    linhaDec+1)+ "  declara   o",p)
126     else:
127         if p.parser.current_func=="0": #est  a ser redeclarada globalmente
128             error("Vari vel global "+varName+" foi redeclarada na linha"+str(p.
                    parser.func_var_counter.get(p.parser.current_func)+1)+"das
                    declara   es depois de ter sido declarada na linha"+str(linhaDec+1),p
                    )
129         else: #est  a ser redeclarada localmente
130             error("Vari vel "+varName+" redeclarada na fun   o "+p.parser.
                    current_func+" como "+str(p.parser.func_var_counter.get(p.parser.
```

```
                current_func))+" declara o depois de ter sido declarada como"+str
                (linhaDec+1)+" ",p)
131
132
133 def p_Declaracao_var_simple(p):
134     "Declaracao : Type ID ';'"
135     varName = p[2].strip()
136     addVar(p,varName,1)
137     if p.parser.current_type=="":
138         p.parser.compiled+= "pushi 0\n"
139     elif p.parser.current_type=="f":
140         p.parser.compiled+= "pushf 0.0\n"
141     elif p.parser.current_type=="s":
142         p.parser.compiled+= "pushs \"\"\n"
143     p.parser.current_type=None
144
145 def p_Declaracao_var_complex(p):
146     "Declaracao : Type ID '=' Expr ';'"
147     varName = p[2].strip()
148     addVar(p,varName,1)
149     p.parser.compiled+=p[4]
150     p.parser.current_type=None
151
152 def p_Declaracao_array(p):
153     "Declaracao : Type Array ID ';'"
154     varName = p[3].strip()
155     array=p[2].strip().split(":")
156     lines = int(array[0])
157     columns = int(array[1])
158     s = p.parser.func_array_info.get(p.parser.current_func)
159     if s is None:
160         p.parser.func_array_info.update({p.parser.current_func : {}})
161     p.parser.func_array_info[p.parser.current_func][varName]=[lines,columns]
162     addVar(p,varName,lines*columns)
163     p.parser.current_type=None
164
165 def p_Type(p):
166     "Type : ID"
167     if p[1]=="int" :
168         p.parser.current_type=""
169     elif p[1]=="float" :
170         p.parser.current_type="f"
171     elif p[1]=="string" :
172         p.parser.current_type="s"
173     else :
174         error("ERROR: Tipo Desconhecido \""+p[1]+"\"",p)
175
176 def p_Array(p):
177     "Array : '[' INT ']' ContArray"
178     amount = int(p[2]) * int(p[4])
179     p[0] = p[2]+":"+p[4]
180     if p.parser.current_type=="":
181         string = "pushi 0\n"
182     elif p.parser.current_type=="f":
```

```
183             string = "pushf 0.0\n"
184         elif p.parser.current_type=="s":
185             string="pushs \"\"\n"
186         while amount>0:
187             p.parser.compiled+=string
188             amount-=1
189 
190 def p_ContArray(p):
191     "ContArray : '[' INT ']'"
192     p[0] = p[2]
193 
194 def p_ContArray_paragem(p) :
195     "ContArray : "
196     p[0] = "1"
197 
198 def p_Expr_add(p):
199     "Expr : Expr '+' Term"
200     p[0] = p[3] + p[1]
201     if p.parser.current_type=="s":
202         p[0]+= "concat\n"
203     else:
204         p[0]+= p.parser.current_type + "add\n"
205 
206 def p_Expr_sub(p):
207     "Expr : Expr '-' Term"
208     if p.parser.current_type=="s":
209         error("ERRO: A opera   o '-' n o pode ser utilizada para strings",p)
210     else:
211         p[0] = p[1] + p[3] + p.parser.current_type + "sub\n"
212 
213 def p_Expr_paragem(p):
214     "Expr : Term"
215     p[0] = p[1]
216 
217 def p_Term_mul(p):
218     "Term : Term '*' Factor"
219     if p.parser.current_type=="s":
220         error("ERRO: A opera   o '/' n o pode ser utilizada para strings",p)
221     else:
222         p[0] = p[1] + p[3] + p.parser.current_type + "mul\n"
223 
224 def p_Term_div(p):
225     "Term : Term '/' Factor"
226     if p.parser.current_type=="s":
227         error("ERRO: A opera   o '/' n o pode ser utilizada para strings",p)
228     else:
229         p[0] = p[1] + p[3] + p.parser.current_type + "div\n"
230 
231 def p_Term_mod(p):
232     "Term : Term '%' Factor"
233     if p.parser.current_type=="f":
234         error("ERRO: A opera   o % n o pode ser utilizada para floats", p)
235     elif p.parser.current_type=="s":
236         error("ERRO: A opera   o % n o pode ser utilizada para strings", p)
```

```
237     else:
238         p[0] = p[1] + p[3] + "mod\n"
239
240 def p_Term_paragem(p):
241     "Term : Factor"
242     p[0] = p[1]
243
244 def convertType(p,s):
245     r = ""
246     if s=="f":
247         if p.parser.current_type is None :
248             p.parser.current_type="f"
249         elif p.parser.current_type=="":
250             r = "ftoi\n"
251         elif p.parser.current_type=="s":
252             r = "strf\n"
253     elif s=="":
254         if p.parser.current_type is None:
255             p.parser.current_type=""
256         elif p.parser.current_type=="f" :
257             r = "itof\n"
258         elif p.parser.current_type=="s":
259             r = "stri\n"
260     elif s=="s":
261         if p.parser.current_type is None:
262             p.parser.current_type="s"
263         elif p.parser.current_type=="":
264             r = "atoi\n"
265         elif p.parser.current_type=="f":
266             r = "atof\n"
267     return r
268
269 def p_Factor(p):
270     "Factor : INT"
271     p[0] = "pushi " + p[1] + "\n" + convertType(p,"")
272
273 def p_Factor_neg(p):
274     "Factor : '-' INT"
275     p[0] = "pushi " + p[2] + "\n" + "pushi -1\n" + "mul\n" + convertType(p,"")
276
277 def p_FactorF(p):
278     "Factor : FLOAT"
279     p[0] = "pushf " + p[1] + "\n" + convertType(p,"f")
280
281 def p_FactorF_neg(p):
282     "Factor : '-' FLOAT"
283     p[0] = "pushf " + p[2] + "\n" + "pushf -1.0\n" + "fmul\n" + convertType(p,"f")
284
285 def p_FactorS(p):
286     "Factor : STRING"
287     p[0] = "pushs " + p[1] + "\n" + convertType(p,"s")
288
289 def p_Factor_func(p):
290     "Factor : CALL ID"
```

```
291     if p.parser.current_type=="" :
292         p[0]="pushi 0\n"
293     elif p.parser.current_type=="f":
294         p[0]="pushf 0.0\n"
295     elif p.parser.current_type=="s":
296         p[0]="pushs \"\"\n"
297     p[0]+="pusha "+p[2].strip()+"\ncall\n"
298
299 def p_Factor_var_simple(p):
300     "Factor : ID"
301     varName=p[1].strip()
302     k = p.parser.func_vars.get(p.parser.current_func).get(varName)
303     t = p.parser.func_var_types.get(p.parser.current_func).get(varName)
304     if k is None or p.parser.current_func == "0":
305         k = p.parser.func_vars.get("0").get(varName)
306         t = p.parser.func_var_types.get("0").get(varName)
307         if k is None:
308             error("Variavel "+varName+" n o declarada",p)
309         else:
310             p[0] = "pushg " + str(k) + "\n" + convertType(p,t)
311     else:
312         p[0] = "pushl " + str(k) + "\n" + convertType(p,t)
313
314
315 def p_Factor_var_array(p):
316     "Factor : ID '[' INT ']' ArrayCol"
317     varName=p[1].strip()
318     line = int(p[3])
319     column = int(p[5])
320     k = p.parser.func_array_info.get(p.parser.current_func).get(varName)
321     if k is None or p.parser.current_func == "0":
322         k = p.parser.func_array_info.get("0").get(varName)
323     if k is None:
324         error("Array"+p[1]+" n o declarado",p)
325     if line<0:
326         error("Linha acedida no array tem  ndice  negativo",p)
327     if column<0:
328         error("Coluna acedida no array tem  ndice  negativo",p)
329     if k[0]<=line:
330         error("Array"+p[1]+"tem"+k[0]+"linhas e foi acedida a linha n mero "+line-1,
            p)
331     elif k[1]<=column:
332         error("Array"+p[1]+"tem"+k[1]+"colunas e foi acedida a coluna n mero "+
            column-1,p)
333     else:
334         pos = p.parser.func_vars.get(p.parser.current_func).get(varName)
335         t = p.parser.func_var_types.get(p.parser.current_func).get(varName)
336         if pos is None or p.parser.current_func == "0":
337             pos = p.parser.func_vars.get("0").get(varName)
338             t = p.parser.func_var_types.get("0").get(varName)
339             place=pos+line*k[1]+column
340             p[0] = "pushg " + str(place) + "\n" + convertType(p,t)
341         else:
342             place=pos+line*k[1]+column
```

```
343             p[0] = "pushl " + str(place) + "\n" + convertType(p,t)
344
345 def p_ArrayCol_cols(p):
346     "ArrayCol : '[' INT ']'"
347     p[0]=p[2]
348
349 def p_ArrayCol_no_cols(p):
350     "ArrayCol : "
351     p[0]="0"
352
353 def p_Codigo_decFunc(p):
354     "Codigo : Codigo Funcao"
355
356 def p_Codigo_Instrucao(p):
357     "Codigo : Codigo Instrucao"
358
359 def p_Codigo_paragem(p):
360     "Codigo : "
361
362 def p_Funcao(p):
363     "Funcao : DecFunc '{' Variaveis ListInstrucao EndFunc"
364     p.parser.current_func = "0"
365
366 def p_DecFunc(p):
367     "DecFunc : FUNC_ID" #Adiciona todas as informa  es da fun   o para seres
            atualizadas aquando da interpreta   o da mesma
368     p.parser.stack.append(p.parser.label)
369     p.parser.label+=1
370     p.parser.compiled+= "jump af" + str(p.parser.stack[-1]) + "\n" + p[2].strip() + "
            :\n"
371     p.parser.current_func = p[2].strip()
372     p.parser.func_var_counter[p.parser.current_func] = 0
373     p.parser.func_vars[p.parser.current_func]={}
374     p.parser.func_var_types[p.parser.current_func]={}
375
376 def p_EndFunc(p):
377     "EndFunc : '}'"
378     p.parser.compiled+="af" + str(p.parser.stack.pop()) + " :\n"
379     p.parser.current_type=None
380
381 def p_Ret(p):
382     "Ret : RET Expr ';'"
383     if p.parser.current_func=="0":
384         error("Linha de retorno de valor colocada globalmente",p)
385     p.parser.compiled+= p[2] + "storel -1\nreturn\n"
386
387
388 def p_ListInstrucao(p):
389     "ListInstrucao : ListInstrucao Instrucao"
390
391 def p_ListInstrucao_paragem(p):
392     "ListInstrucao : "
393
394 def p_Instrucao_atribuicao(p):
```

```
395     "Instrucao : Atribuicao"
396     p.parser.compiled+=p[1]
397
398 def p_Instrucao_reading(p):
399     "Instrucao : Leitura"
400
401 def p_Instrucao_writing(p):
402     "Instrucao : Escrita"
403
404 def p_Instrucao_condicao(p):
405     "Instrucao : Condicao"
406
407 def p_Instrucao_ciclo(p):
408     "Instrucao : Ciclo"
409
410 def p_Instrucao_return(p):
411     "Instrucao : Ret"
412
413 def atribuiValor(p,array):
414     stringResultante = ""
415     k = p.parser.func_vars.get(p.parser.current_func).get(array[0].strip())
416     v=p.parser.current_func
417     if k is None or p.parser.current_func == "0":
418         v="0"
419         k = p.parser.func_vars.get("0").get(array[0].strip())
420         if k is None:
421             error("Vari vel"+ array[0]+"n o declarada",p)
422         stringResultante+= "storeg "
423     else:
424         stringResultante+= "storel "
425     if len(array)==1:
426         stringResultante+= str(k) + "\n"
427     else:
428         info = p.parser.func_array_info.get(v).get(array[0].strip())
429         if int(info[0])<=int(array[1]):
430             error("Array"+array[0]+"tem"+info[0]+"linhas e foi acedida a linha
                n mero "+str(int(array[1])-1),p)
431         elif int(info[1])<=int(array[2]):
432             error("Array"+array[0]+"tem"+info[1]+"colunas e foi acedida a coluna
                n mero "+str(int(array[2])-1),p)
433         else:
434             stringResultante+=str(k+int(array[1])*int(info[1])+int(array[2])) +"\n"
435     return stringResultante
436
437 def p_Atribuicao(p):
438     "Atribuicao : Var '=' Expr ';'"
439     p[0] = p[3]
440     array=p[1].strip().split(":")
441     p[0]+=atribuiValor(p,array)
442     p.parser.current_type=None
443
444 def p_Var(p):
445     "Var : ID ContinuacaoVar"
446     t = p.parser.func_var_types.get(p.parser.current_func).get(p[1].strip())
```

```
447     if t==None:
448         t = p.parser.func_var_types.get("0").get(p[1].strip())
449     if t is None:
450         error("Vari vel "+p[1]+" n o declarada",p)
451     p.parser.current_type=t
452     p[0]=p[1]+p[2]
453
454 def p_ContinuacaoVar_array(p):
455     "ContinuacaoVar : '[' INT ']' ContContinuacaoVar"
456     p[0]=":"+p[2]+p[4]
457
458 def p_ContinuacaoVar_simples(p):
459     "ContinuacaoVar : "
460     p[0]=""
461
462 def p_ContContinuacaoVar_array_duplo(p):
463     "ContContinuacaoVar : '[' INT ']'"
464     p[0]=":" + p[2]
465
466 def p_ContContinuacaoVar_array_simples(p):
467     "ContContinuacaoVar : "
468     p[0]=":0"
469
470 def p_Leitura(p):
471     "Leitura : READ Var ';'"
472     array = p[2].strip().split(":")
473     t = p.parser.func_var_types.get(p.parser.current_func).get(array[0])
474     if t is None or p.parser.current_func == "0":
475         t = p.parser.func_var_types.get("0").get(array[0])
476     p.parser.compiled+="read\n"
477     p.parser.compiled+="dup 1\n"
478     if t=="" :
479         p.parser.compiled+="atoi\n"
480     elif t=="f":
481         p.parser.compiled+="atof\n"
482     p.parser.compiled+= atribuiValor(p,array) + "free\n"
483     p.parser.current_type=None
484
485 def p_Escrita(p):
486     "Escrita : WRITE Expr ';'"
487     p.parser.compiled+=p[2]
488     if p.parser.current_type=="" :
489         p.parser.compiled+="writei\n"
490     elif p.parser.current_type=="f" :
491         p.parser.compiled+="writef\n"
492     elif p.parser.current_type=="s":
493         p.parser.compiled+="writes\n"
494     p.parser.current_type=None
495
496 def p_Condicao_if(p):
497     "Condicao : IF ExprCond Do '{' ListInstrucao '}' ';'"
498     p.parser.compiled+="e"+str(p.parser.stack.pop()) + " :\n"
499
500 def p_Condicao_if_else(p):
```

```
501     "Condicao : IF ExprCond Do '{' ListInstrucao '}' Else '{' ListInstrucao '}' ';'"
502     p.parser.compiled+="t"+str(p.parser.stack.pop())+" :\n"
503
504 def p_Do(p):
505     "Do : DO"
506     p.parser.stack.append(p.parser.label)
507     p.parser.compiled+="jz e" + str(p.parser.stack[-1]) + "\n"
508     p.parser.label+=1
509     p.parser.current_type=None
510
511 def p_Else(p):
512     "Else : ELSE"
513     p.parser.compiled+="jump t" + str(p.parser.stack[-1]) +"\n"+"e"+str(p.parser.
            stack[-1]) + " :\n"
514
515 def p_ExprCond_and(p):
516     "ExprCond : ExprCond '&' Cond"
517     p.parser.compiled+="mul\n"
518
519 def p_ExprCond_or(p):
520     "ExprCond : ExprCond '|' Cond"
521     p.parser.compiled+="add\n"
522
523 def p_ExprCond_paragem(p):
524     "ExprCond : Cond"
525
526 def p_Cond_equals(p):
527     "Cond : Expr EQUAL Expr"
528     p.parser.compiled+= p[1] + p[3] + "equal\n"
529
530 def p_Cond_not_equals(p):
531     "Cond : Expr NOTEQUAL Expr"
532     p.parser.compiled+= p[1] + p[3] + "equal\nnot\n"
533
534 def p_Cond_bigger(p):
535     "Cond : Expr BIGGER Expr"
536     p.parser.compiled+= p[1] + p[3] + "sup\n"
537
538 def p_Cond_smaller(p):
539     "Cond : Expr SMALLER Expr"
540     p.parser.compiled+= p[1] + p[3] + "inf\n"
541
542 def p_Cond_biggerequal(p):
543     "Cond : Expr BIGGEREQUAL Expr"
544     p.parser.compiled+= p[1] + p[3] + "supeq\n"
545
546 def p_Cond_smallerequal(p):
547     "Cond : Expr SMALLEREQUAL Expr"
548     p.parser.compiled+= p[1] + p[3] + "infeq\n"
549
550 def p_Ciclo_while(p):
551     "Ciclo : CicloWhile"
552
553 def p_Ciclo_repeat(p):
```

```
554     "Ciclo : CicloRepeat"
555
556 def p_Ciclo_for(p):
557     "Ciclo : CicloFor"
558
559 def p_CicloWhile(p):
560     "CicloWhile : While ExprCond Do '{' ListInstrucao '}'"
561     pop1 = p.parser.stack.pop()
562     pop2 = p.parser.stack.pop()
563     p.parser.compiled+="jump c" + str(pop2) + "\n" + "e"+str(pop1) + " :\n"
564
565 def p_While(p):
566     "While : WHILE"
567     p.parser.stack.append(p.parser.label)
568     p.parser.compiled+="c"+str(p.parser.stack[-1]) +" :\n"
569     p.parser.label+=1
570
571 def p_CicloRepeat(p):
572     "CicloRepeat : Repeat ListInstrucao Until ExprCond"
573     p.parser.compiled+="jz c" + str(p[3]) + "\n"
574
575 def p_Repeat(p):
576     "Repeat : REPEAT"
577     p.parser.stack.append(p.parser.label)
578     p.parser.compiled+="jump u" + str(p.parser.stack[-1]) +"\n" + "c" + str(p.parser.
            stack[-1]) + " :\n"
579     p.parser.label+=1
580     p.parser.current_type = None
581
582 def p_Until(p):
583     "Until : UNTIL"
584     p[0] = p.parser.stack.pop()
585     p.parser.compiled+= "u"+str(p[0]) + " :\n"
586
587 def p_CicloFor(p):
588     "CicloFor : FOR '(' AtribuicaoInicial ExprCond EndCond Atribuicao ')' '{'
            ListInstrucao '}'"
589     pop1 = p.parser.stack.pop()
590     pop2 = p.parser.stack.pop()
591     p.parser.compiled+=p[6]+"jump f" + str(pop2) + "\nff"+str(pop1) +" :\n"
592
593 def p_AtribuicaoInicial(p):
594     "AtribuicaoInicial : Atribuicao"
595     p.parser.stack.append(p.parser.label)
596     p.parser.compiled+=p[1]+"f"+ str(p.parser.stack[-1]) +" :\n"
597     p.parser.label+=1
598
599 def p_EndCond(p):
600     "EndCond : ';'"
601     p.parser.stack.append(p.parser.label)
602     p.parser.compiled+="jz ff"+str(p.parser.stack[-1]) + "\n"
603     p.parser.label+=1
604
605 def p_error(p):
```

```
606     print("Erro sint tico: ", p) #Imprime erros de gram tica
607     parser.success = False
608
609 def error(arg0, p): #Imprime erros de compila  o
610     print(arg0)
611     p.parser.success=False
612     p[0]="0"
613
614 # Build the parser
615
616 parser = yacc.yacc()
617
618 # Read input and parse it by line
619
620 import sys
621
622 if len(sys.argv)==1 : #Verifica se foi fornecido um ficheiro para compilar
623     print("Nenhum ficheiro escolhido para compilar")
624 else :
625     print("Compiling: ",sys.argv[1]) # Imprime ficheiro de onde est  a ler
626
627 read = open(sys.argv[1], "r")
628
629 if len(sys.argv)==2 : #Verifica se foi fornecido um ficheiro destino para o c digo
    pseudo-m quina
630     filename = "a.vm"
631 else :
632     filename = sys.argv[2]
633
634 parser.success=True #Determina se a compila  o foi bem sucedida
635
636 parser.compiled = "" #Texto compilado guarda-se na string para escrever no fim da
    compila  o caso tenha
637                      #sido successful
638
639 parser.current_type=None #Determina o tipo da opera  o atual, pode ser ""(Inteiro)
    "f"(Float) "s"(String)
640                          #None(N o definido)
641
642 parser.current_func="0" #Determina a fun  o onde se encontra atualmente de modo a
    poder produzir melhores
643                         #mensagens de erro e controlar as declara  es locais ("0"
                                significa global)
644
645 parser.func_var_counter={} #Guarda o contador de vari veis de cada fun  o
646 parser.func_var_counter.update({"0" : 0})
647
648 parser.func_vars={} #Guarda todas as vari veis das fun  es declaradas
649 parser.func_vars.update({"0" : {}})
650
651 parser.func_var_types={} #Guarda todos os tipos das vari veis de fun  es
    declaradas
652 parser.func_var_types.update({"0" : {}})
653
```

```
654 parser.func_array_info={} #Guarda um map para cada fun   o com um tuplo de linhas e
        colunas que os arrays nessa
655                           #fun   o declarados t m
656 parser.func_array_info.update({"0" : {}})
657
658 parser.label=0 #Serve para declarar as etiquetas dos ciclos, fun   es e condi   es
659
660 parser.stack=[]
661
662 content=""
663
664 for linha in read:
665     if not(linha.strip().startswith('#')) : #Remove qualquer linha come ada por #
            pois estas linhas s o de
666                                             #coment rio
667         content += linha
668 parser.parse(content)
669
670 if parser.success:
671     print("Compila   o bem sucedida")
672     print("Creating: ",filename)
673     write = open(filename, "w+")
674     write.write(parser.compiled)
675 else:
676     print("Compila   o mal sucedida")
```

```
'''
T:{ID,INT,FLOAT,STRING,READ,WRITE,EQUAL,BIGGER,BIGGEREQUAL,SMALLER,SMALLEREQUAL,RET,IF,DO,ELSE,
    WHILE,FUNC,CALL,REPEAT,UNTIL,FOR,';','=','+','-','*','/','%','{','}','&','|',':','[',']'}

N:{Inicio,Start,Variaveis,Codigo,Declaracoes,Declaracao,
    Type,Array,ContArray,Expr,Term,Factor,Funcao,
    DecFunc,Ret,ListInstrucao,Instrucao,Atribuicao,Var,
    ContContinuacaoVar,Leitura,Escrita,Condicao,Ciclo,
    ExprCond,Do,Else,Cond,Ciclo,CicloWhile,While,Repeat,Until,
    CicloRepeat,CicloFor,AtribuicaoInicial}

S: Inicio

P:{
    p1: Inicio => Start Variaveis Codigo
    p2: Start =>
    p3: Variaveis => '{' Declaracoes '}'
    p4: Declaracoes => Declaracoes Declaracao
    p5:             |
    p6: Declaracao => Type ID ';'
    p7:            | '=' Expr ';'
    p8:            | Array ID ';'
    p9: Type => ID
    p10: Array => '[' INT ']' '[' INT ']'
    p11:       |  '[' INT ']'
    p12: Expr => Expr '+' Term
    p13:      |  Expr '-' Term
    p14:      | Term
    p15: Term => Term '*' Factor
    p16:      |  Term '/' Factor
    p17:      |  Term '%' Factor
    p18:      |  Factor
    p19: Factor => INT
    p20:        | '-' INT
    p21:        | FLOAT
    p22:        | '-' FLOAT
    p23:        | ID
    p24:        | ID '[' INT ']'
    p25:        | ID '[' INT ']' '[' INT ']'
    p26:        | CALL ID
    p27: Codigo => Funcao Codigo
    p28:        | Instrucao Codigo
    p29:        |
    p30: Funcao => DecFunc '{' Variaveis ListInstrucao Ret '}'
    p31: DecFunc => FUNC ID
    p32: Ret => RET Expr ';'
    p33: ListInstrucao => ListInstrucao Instrucao
    p34:               |
    p35: Instrucao => Atribuicao
    p36:           | Leitura
    p37:           | Escrita
    p38:           | Condicao
    p39:           | Ciclo
```

Figura B.1: Parte 1

```
    p40: Atribuicao => Var '=' Expr ';'
    p41: Var => ID
    p42:     |  ID '[' INT ']'
    p43:     |  ID '[' INT ']' '[' INT ']'
    p44: Leitura => READ Var ';'
    p45: Escrita => WRITE Var ';'
    p46: Condicao => IF ExprCond Do '{' ListInstrucao '}' ';'
    p47:          |  IF ExprCond Do '{' ListInstrucao '}' Else '{' ListInstrucao '}' ';'
    p48: Do => DO
    p49: Else => ELSE
    p50: ExprCond => ExprCond '&' Cond
    p51:          |  ExprCond '|' Cond
    p52:          |
    p53: Cond => Expr EQUAL Expr
    p54:      |  Expr BIGGER Expr
    p55:      |  Expr BIGGEREQUAL Expr
    p56:      |  Expr SMALLER Expr
    p57:      |  Expr SMALLEREQUAL Expr
    p58: Ciclo => CicloWhile
    p59:       | CicloRepeat
    p60:       | CicloFor
    p61: CicloWhile => While ExprCond Do '{' ListInstrucao '}'
    p62: While => WHILE
    p63: CicloRepeat => Repeat '{' ListInstrucao '}' Until ExprCond
    p64: Repeat => REPEAT
    p65: Until => UNTIL
    p66: CicloFor => FOR '(' AtribuicaoInicial ExprCond EndCond Atribuicao ')' '{' ListInstrucao '}'
    p67: AtribuicaoInicial => Atribuicao
    p68: EndCond => ';'
}
'''
import ply.yacc as yacc

from interpretador_lex import tokens,literals

def p_Inicio(p):
    "Inicio : Variaveis Start Codigo"

def p_Start(p):
    "Start : "
    p.parser.compiled+="start\n"

def p_Variaveis(p):
    "Variaveis : '{' Declaracoes '}'"

def p_Declaracoes(p):
    "Declaracoes : Declaracoes Declaracao"

def p_Declaracoes_paragem(p):
    "Declaracoes : "

def getAndIncFuncVarCounter(p,inc):
    varNumAtual = p.parser.func_var_counter.get(p.parser.current_func)
    p.parser.func_var_counter[p.parser.current_func] = varNumAtual + inc
    return varNumAtual
```

Figura B.2: Parte 2

```
def addVarToFunc(p,varname,inc):
    p.parser.func_vars[p.parser.current_func][varname] = getAndIncFuncVarCounter(p,inc)
    p.parser.func_var_types[p.parser.current_func][varname] = p.parser.current_type

def addVar(p,varName,inc):
    linhaDec = p.parser.func_vars.get(p.parser.current_func).get(varName)
    if linhaDec == None: #não foi declarado na função atual
        linhaDec = p.parser.func_vars.get("0").get(varName) #pode ter sido declarada globalmente
        if linhaDec is None: #não foi declarada globalmente nem localmente logo pode ser adicionada
            addVarToFunc(p,varName,inc)
        else: #foi declarada globalmente e está a ser redeclarada localmente
            p.parser.success=False
            error("Variável "+varName+" declarada localmente na função "+p.parser.current_func+"
depois de ter sido declarada globalmente na "+str(linhaDec+1)+ "ª declaração",p)
    else:
        if p.parser.current_func=="0": #está a ser redeclarada globalmente
            error("Variável global "+varName+" foi redeclarada na
linha"+str(p.parser.func_var_counter.get(p.parser.current_func)+1)+"das declarações depois de ter sido
declarada na linha"+str(linhaDec+1),p)
        else: #está a ser redeclarada localmente
            error("Variável "+varName+" redeclarada na função "+p.parser.current_func+" como
"+str(p.parser.func_var_counter.get(p.parser.current_func))+"ª declaração depois de ter sido declarada
como"+str(linhaDec+1)+"ª",p)


def p_Declaracao_var_simple(p):
    "Declaracao : Type ID ';'"
    varName = p[2].strip()
    addVar(p,varName,1)
    if p.parser.current_type=="":
        p.parser.compiled+= "pushi 0\n"
    elif p.parser.current_type=="f":
        p.parser.compiled+= "pushf 0.0\n"
    elif p.parser.current_type=="s":
        p.parser.compiled+= "pushs \"\"\n"
    p.parser.current_type=None

def p_Declaracao_var_complex(p):
    "Declaracao : Type ID '=' Expr ';'"
    varName = p[2].strip()
    addVar(p,varName,1)
    p.parser.compiled+=p[4]
    p.parser.current_type=None

def p_Declaracao_array(p):
    "Declaracao : Type Array ID ';'"
    varName = p[3].strip()
    array=p[2].strip().split(":")
    lines = int(array[0])
    columns = int(array[1])
    s = p.parser.func_array_info.get(p.parser.current_func)
    if s is None:
        p.parser.func_array_info.update({p.parser.current_func : {}})
    p.parser.func_array_info[p.parser.current_func][varName]=[lines,columns]
    addVar(p,varName,lines*columns)
```

Figura B.3: Parte 3

```
    p.parser.current_type=None

def p_Type(p):
    "Type : ID"
    if p[1]=="int" :
        p.parser.current_type=""
    elif p[1]=="float" :
        p.parser.current_type="f"
    elif p[1]=="string" :
        p.parser.current_type="s"
    else :
        error("ERROR: Tipo Desconhecido \""+p[1]+"\"",p)

def p_Array(p):
    "Array : '[' INT ']' ContArray"
    amount = int(p[2]) * int(p[4])
    p[0] = p[2]+":"+p[4]
    if p.parser.current_type=="":
        string = "pushi 0\n"
    elif p.parser.current_type=="f":
        string = "pushf 0.0\n"
    elif p.parser.current_type=="s":
        string="pushs \"\"\n"
    while amount>0:
        p.parser.compiled+=string
        amount-=1

def p_ContArray(p):
    "ContArray : '[' INT ']'"
    p[0] = p[2]

def p_ContArray_paragem(p) :
    "ContArray : "
    p[0] = "1"

def p_Expr_add(p):
    "Expr : Expr '+' Term"
    p[0] = p[3] + p[1]
    if p.parser.current_type=="s":
        p[0]+= "concat\n"
    else:
        p[0]+= p.parser.current_type + "add\n"

def p_Expr_sub(p):
    "Expr : Expr '-' Term"
    if p.parser.current_type=="s":
        error("ERRO: A operação '-' não pode ser utilizada para strings",p)
    else:
        p[0] = p[1] + p[3] + p.parser.current_type + "sub\n"

def p_Expr_paragem(p):
    "Expr : Term"
    p[0] = p[1]

def p_Term_mul(p):
```

Figura B.4: Parte 4

```
    "Term : Term '*' Factor"
    if p.parser.current_type=="s":
        error("ERRO: A operação '/' não pode ser utilizada para strings",p)
    else:
        p[0] = p[1] + p[3] + p.parser.current_type + "mul\n"

def p_Term_div(p):
    "Term : Term '/' Factor"
    if p.parser.current_type=="s":
        error("ERRO: A operação '/' não pode ser utilizada para strings",p)
    else:
        p[0] = p[1] + p[3] + p.parser.current_type + "div\n"

def p_Term_mod(p):
    "Term : Term '%' Factor"
    if p.parser.current_type=="f":
        error("ERRO: A operação % não pode ser utilizada para floats", p)
    elif p.parser.current_type=="s":
        error("ERRO: A operação % não pode ser utilizada para strings", p)
    else:
        p[0] = p[1] + p[3] + "mod\n"

def p_Term_paragem(p):
    "Term : Factor"
    p[0] = p[1]

def convertType(p,s):
    r = ""
    if s=="f":
        if p.parser.current_type is None :
            p.parser.current_type="f"
        elif p.parser.current_type=="":
            r = "ftoi\n"
        elif p.parser.current_type=="s":
            r = "strf\n"
    elif s=="":
        if p.parser.current_type is None:
            p.parser.current_type=""
        elif p.parser.current_type=="f" :
            r = "itof\n"
        elif p.parser.current_type=="s":
            r = "stri\n"
    elif s=="s":
        if p.parser.current_type is None:
            p.parser.current_type="s"
        elif p.parser.current_type=="":
            r = "atoi\n"
        elif p.parser.current_type=="f":
            r = "atof\n"
    return r

def p_Factor(p):
    "Factor : INT"
    p[0] = "pushi " + p[1] + "\n" + convertType(p,"")

def p_Factor_neg(p):
```

Figura B.5: Parte 5

```
    "Factor : '-' INT"
    p[0] = "pushi " + p[2] + "\n" + "pushi -1\n" + "mul\n" + convertType(p,"")

def p_FactorF(p):
    "Factor : FLOAT"
    p[0] = "pushf " + p[1] + "\n" + convertType(p,"f")

def p_FactorF_neg(p):
    "Factor : '-' FLOAT"
    p[0] = "pushf " + p[2] + "\n" + "pushf -1.0\n" + "fmul\n" + convertType(p,"f")

def p_FactorS(p):
    "Factor : STRING"
    p[0] = "pushs " + p[1] + "\n" + convertType(p,"s")

def p_Factor_func(p):
    "Factor : CALL ID"
    if p.parser.current_type=="" :
        p[0]="pushi 0\n"
    elif p.parser.current_type=="f":
        p[0]="pushf 0.0\n"
    elif p.parser.current_type=="s":
        p[0]="pushs \"\"\n"
    p[0]+="pusha "+p[2].strip()+"\ncall\n"

def p_Factor_var_simple(p):
    "Factor : ID"
    varName=p[1].strip()
    k = p.parser.func_vars.get(p.parser.current_func).get(varName)
    t = p.parser.func_var_types.get(p.parser.current_func).get(varName)
    if k is None or p.parser.current_func == "0":
        k = p.parser.func_vars.get("0").get(varName)
        t = p.parser.func_var_types.get("0").get(varName)
        if k is None:
            error("Variavel "+varName+" não declarada",p)
        else:
            p[0] = "pushg " + str(k) + "\n" + convertType(p,t)
    else:
        p[0] = "pushl " + str(k) + "\n" + convertType(p,t)


def p_Factor_var_array(p):
    "Factor : ID '[' INT ']' ArrayCol"
    varName=p[1].strip()
    line = int(p[3])
    column = int(p[5])
    k = p.parser.func_array_info.get(p.parser.current_func).get(varName)
    if k is None or p.parser.current_func == "0":
        k = p.parser.func_array_info.get("0").get(varName)
    if k is None:
        error("Array"+p[1]+"não declarado",p)
    if line<0:
        error("Linha acedida no array tem índice negativo",p)
    if column<0:
        error("Coluna acedida no array tem índice negativo",p)
    if k[0]<=line:
```

Figura B.6: Parte 6

```
        if k[0]<=line:
            error("Array"+p[1]+"tem"+k[0]+"linhas e foi acedida a linha número "+line-1,p)
        elif k[1]<=column:
            error("Array"+p[1]+"tem"+k[1]+"colunas e foi acedida a coluna número "+column-1,p)
        else:
            pos = p.parser.func_vars.get(p.parser.current_func).get(varName)
            t = p.parser.func_var_types.get(p.parser.current_func).get(varName)
            if pos is None or p.parser.current_func == "0":
                pos = p.parser.func_vars.get("0").get(varName)
                t = p.parser.func_var_types.get("0").get(varName)
                place=pos+line*k[1]+column
                p[0] = "pushg " + str(place) + "\n" + convertType(p,t)
            else:
                place=pos+line*k[1]+column
                p[0] = "pushl " + str(place) + "\n" + convertType(p,t)

def p_ArrayCol_cols(p):
    "ArrayCol : '[' INT ']'"
    p[0]=p[2]

def p_ArrayCol_no_cols(p):
    "ArrayCol : "
    p[0]="0"

def p_Codigo_decFunc(p):
    "Codigo : Codigo Funcao"

def p_Codigo_Instrucao(p):
    "Codigo : Codigo Instrucao"

def p_Codigo_paragem(p):
    "Codigo : "

def p_Funcao(p):
    "Funcao : DecFunc '{' Variaveis ListInstrucao EndFunc"
    p.parser.current_func = "0"

def p_DecFunc(p):
    "DecFunc : FUNC ID" #Adiciona todas as informações da função para seres atualizadas aquando da
interpretação da mesma
    p.parser.stack.append(p.parser.label)
    p.parser.label+=1
    p.parser.compiled+= "jump af" + str(p.parser.stack[-1]) + "\n" + p[2].strip() + " :\n"
    p.parser.current_func = p[2].strip()
    p.parser.func_var_counter[p.parser.current_func] = 0
    p.parser.func_vars[p.parser.current_func]={}
    p.parser.func_var_types[p.parser.current_func]={}

def p_EndFunc(p):
    "EndFunc : '}'"
    p.parser.compiled+="af" + str(p.parser.stack.pop()) + " :\n"
    p.parser.current_type=None

def p_Ret(p):
    "Ret : RET Expr ';'"
    if p.parser.current_func=="0":
```

Figura B.7: Parte 7

```
        error("Linha de retorno de valor colocada globalmente",p)
    p.parser.compiled+= p[2] + "storel -1\nreturn\n"


def p_ListInstrucao(p):
    "ListInstrucao : ListInstrucao Instrucao"

def p_ListInstrucao_paragem(p):
    "ListInstrucao : "

def p_Instrucao_atribuicao(p):
    "Instrucao : Atribuicao"
    p.parser.compiled+=p[1]

def p_Instrucao_reading(p):
    "Instrucao : Leitura"

def p_Instrucao_writing(p):
    "Instrucao : Escrita"

def p_Instrucao_condicao(p):
    "Instrucao : Condicao"

def p_Instrucao_ciclo(p):
    "Instrucao : Ciclo"

def p_Instrucao_return(p):
    "Instrucao : Ret"

def atribuiValor(p,array):
    stringResultante = ""
    k = p.parser.func_vars.get(p.parser.current_func).get(array[0].strip())
    v=p.parser.current_func
    if k is None or p.parser.current_func == "0":
        v="0"
        k = p.parser.func_vars.get("0").get(array[0].strip())
        if k is None:
            error("Variável"+ array[0]+"não declarada",p)
        stringResultante+= "storeg "
    else:
        stringResultante+= "storel "
    if len(array)==1:
        stringResultante+= str(k) + "\n"
    else:
        info = p.parser.func_array_info.get(v).get(array[0].strip())
        if int(info[0])<=int(array[1]):
            error("Array"+array[0]+"tem"+info[0]+"linhas e foi acedida a linha número
"+str(int(array[1])-1),p)
        elif int(info[1])<=int(array[2]):
            error("Array"+array[0]+"tem"+info[1]+"colunas e foi acedida a coluna número
"+str(int(array[2])-1),p)
        else:
            stringResultante+=str(k+int(array[1])*int(info[1])+int(array[2])) +"\n"
    return stringResultante

def p_Atribuicao(p):
```

Figura B.8: Parte 8

```
def p_Atribuicao(p):
    "Atribuicao : Var '=' Expr ';'"
    p[0] = p[3]
    array=p[1].strip().split(":")
    p[0]+=atribuiValor(p,array)
    p.parser.current_type=None

def p_Var(p):
    "Var : ID ContinuacaoVar"
    t = p.parser.func_var_types.get(p.parser.current_func).get(p[1].strip())
    if t==None:
        t = p.parser.func_var_types.get("0").get(p[1].strip())
    if t is None:
        error("Variável "+p[1]+" não declarada",p)
    p.parser.current_type=t
    p[0]=p[1]+p[2]

def p_ContinuacaoVar_array(p):
    "ContinuacaoVar : '[' INT ']' ContContinuacaoVar"
    p[0]=":"+p[2]+p[4]

def p_ContinuacaoVar_simples(p):
    "ContinuacaoVar : "
    p[0]=""

def p_ContContinuacaoVar_array_duplo(p):
    "ContContinuacaoVar : '[' INT ']'"
    p[0]=":" + p[2]

def p_ContContinuacaoVar_array_simples(p):
    "ContContinuacaoVar : "
    p[0]=":0"

def p_Leitura(p):
    "Leitura : READ Var ';'"
    array = p[2].strip().split(":")
    t = p.parser.func_var_types.get(p.parser.current_func).get(array[0])
    if t is None or p.parser.current_func == "0":
        t = p.parser.func_var_types.get("0").get(array[0])
    p.parser.compiled+="read\n"
    p.parser.compiled+="dup 1\n"
    if t=="" :
        p.parser.compiled+="atoi\n"
    elif t=="f":
        p.parser.compiled+="atof\n"
    p.parser.compiled+= atribuiValor(p,array) + "free\n"
    p.parser.current_type=None

def p_Escrita(p):
    "Escrita : WRITE Expr ';'"
    p.parser.compiled+=p[2]
    if p.parser.current_type=="" :
        p.parser.compiled+="writei\n"
```

Figura B.9: Parte 9

```
    elif p.parser.current_type=="f" :
        p.parser.compiled+="writef\n"
    elif p.parser.current_type=="s":
        p.parser.compiled+="writes\n"
    p.parser.current_type=None

def p_Condicao_if(p):
    "Condicao : IF ExprCond Do '{' ListInstrucao '}' ';'"
    p.parser.compiled+="e"+str(p.parser.stack.pop()) + " :\n"

def p_Condicao_if_else(p):
    "Condicao : IF ExprCond Do '{' ListInstrucao '}' Else '{' ListInstrucao '}' ';'"
    p.parser.compiled+="t"+str(p.parser.stack.pop())+" :\n"

def p_Do(p):
    "Do : DO"
    p.parser.stack.append(p.parser.label)
    p.parser.compiled+="jz e" + str(p.parser.stack[-1]) + "\n"
    p.parser.label+=1
    p.parser.current_type=None

def p_Else(p):
    "Else : ELSE"
    p.parser.compiled+="jump t" + str(p.parser.stack[-1]) +"\n"+"e"+str(p.parser.stack[-1]) + " :\n"

def p_ExprCond_and(p):
    "ExprCond : ExprCond '&' Cond"
    p.parser.compiled+="mul\n"

def p_ExprCond_or(p):
    "ExprCond : ExprCond '|' Cond"
    p.parser.compiled+="add\n"

def p_ExprCond_paragem(p):
    "ExprCond : Cond"

def p_Cond_equals(p):
    "Cond : Expr EQUAL Expr"
    p.parser.compiled+= p[1] + p[3] + "equal\n"

def p_Cond_not_equals(p):
    "Cond : Expr NOTEQUAL Expr"
    p.parser.compiled+= p[1] + p[3] + "equal\nnot\n"

def p_Cond_bigger(p):
    "Cond : Expr BIGGER Expr"
    p.parser.compiled+= p[1] + p[3] + "sup\n"

def p_Cond_smaller(p):
    "Cond : Expr SMALLER Expr"
    p.parser.compiled+= p[1] + p[3] + "inf\n"

def p_Cond_biggerequal(p):
    "Cond : Expr BIGGEREQUAL Expr"
    p.parser.compiled+= p[1] + p[3] + "supeq\n"
```

Figura B.10: Parte 10

```
def p_Cond_smallerequal(p):
    "Cond : Expr SMALLEREQUAL Expr"
    p.parser.compiled+= p[1] + p[3] + "infeq\n"

def p_Ciclo_while(p):
    "Ciclo : CicloWhile"

def p_Ciclo_repeat(p):
    "Ciclo : CicloRepeat"

def p_Ciclo_for(p):
    "Ciclo : CicloFor"

def p_CicloWhile(p):
    "CicloWhile : While ExprCond Do '{' ListInstrucao '}'"
    pop1 = p.parser.stack.pop()
    pop2 = p.parser.stack.pop()
    p.parser.compiled+="jump c" + str(pop2) + "\n" + "e"+str(pop1) + " :\n"

def p_While(p):
    "While : WHILE"
    p.parser.stack.append(p.parser.label)
    p.parser.compiled+="c"+str(p.parser.stack[-1]) +" :\n"
    p.parser.label+=1

def p_CicloRepeat(p):
    "CicloRepeat : Repeat ListInstrucao Until ExprCond"
    p.parser.compiled+="jz c" + str(p[3]) + "\n"

def p_Repeat(p):
    "Repeat : REPEAT"
    p.parser.stack.append(p.parser.label)
    p.parser.compiled+="jump u" + str(p.parser.stack[-1]) +"\n" + "c" + str(p.parser.stack[-1]) + ": \n"
    p.parser.label+=1
    p.parser.current_type = None

def p_Until(p):
    "Until : UNTIL"
    p[0] = p.parser.stack.pop()
    p.parser.compiled+= "u"+str(p[0]) + " :\n"

def p_CicloFor(p):
    "CicloFor : FOR '(' AtribuicaoInicial ExprCond EndCond Atribuicao ')' '{' ListInstrucao '}'"
    pop1 = p.parser.stack.pop()
    pop2 = p.parser.stack.pop()
    p.parser.compiled+=p[6]+"jump f" + str(pop2) + "\nff"+str(pop1) +" :\n"

def p_AtribuicaoInicial(p):
    "AtribuicaoInicial : Atribuicao"
    p.parser.stack.append(p.parser.label)
    p.parser.compiled+=p[1]+"f"+ str(p.parser.stack[-1]) +" :\n"
    p.parser.label+=1

def p_EndCond(p):
    "EndCond : ';'"
```

Figura B.11: Parte 11

```
    p.parser.stack.append(p.parser.label)
    p.parser.compiled+="jz ff"+str(p.parser.stack[-1]) + "\n"
    p.parser.label+=1

def p_error(p):
    print("Erro sintático: ", p) #Imprime erros de gramática
    parser.success = False

def error(arg0, p): #Imprime erros de compilação
    print(arg0)
    p.parser.success=False
    p[0]="0"

# Build the parser

parser = yacc.yacc()

# Read input and parse it by line

import sys

if len(sys.argv)==1 : #Verifica se foi fornecido um ficheiro para compilar
    print("Nenhum ficheiro escolhido para compilar")
else :
    print("Compiling: ",sys.argv[1]) # Imprime ficheiro de onde está a ler

read = open(sys.argv[1], "r")

if len(sys.argv)==2 : #Verifica se foi fornecido um ficheiro destino para o código pseudo-máquina
    filename = "a.vm"
else :
    filename = sys.argv[2]

parser.success=True #Determina se a compilação foi bem sucedida

parser.compiled = "" #Texto compilado guarda-se na string para escrever no fim da compilação caso tenha
                     #sido successful

parser.current_type=None #Determina o tipo da operação atual, pode ser ""(Inteiro) "f"(Float) "s"(String)
                         #None(Não definido)

parser.current_func="0" #Determina a função onde se encontra atualmente de modo a poder produzir melhores
                        #mensagens de erro e controlar as declarações locais ("0" significa global)

parser.func_var_counter={} #Guarda o contador de variáveis de cada função
parser.func_var_counter.update({"0" : 0})

parser.func_vars={} #Guarda todas as variáveis das funções declaradas
parser.func_vars.update({"0" : {}})

parser.func_var_types={} #Guarda todos os tipos das variáveis de funções declaradas
parser.func_var_types.update({"0" : {}})

parser.func_array_info={} #Guarda um map para cada função com um tuplo de linhas e colunas que os arrays ne
                          #função declarados têm
```

Figura B.12: Parte 12

```
parser.func_array_info.update({"0" : {}})

parser.label=0 #Serve para declarar as etiquetas dos ciclos, funções e condições

parser.stack=[]

content=""

for linha in read:
    if not(linha.strip().startswith('#')) : #Remove qualquer linha começada por # pois estas linhas são de
                                            #comentário
        content += linha
parser.parse(content)

if parser.success:
    print("Compilação bem sucedida")
    print("Creating: ",filename)
    write = open(filename, "w+")
    write.write(parser.compiled)
else:
    print("Compilação mal sucedida")
```

Figura B.13: Parte 13

Apêndice C

Exemplo de ficheiros de código Joma

C.1 Programa para o simples de cálculo de uma potência

C.1.1 Ficheiro .jm

Listing C.1: Programa em código Joma para calcular uma potência e indicar o tamanho do valor comparando-o com 50 e 100

```
1 {
2 string ret;
3 }
4
5 <func> potencia {
6     {
7         float base;
8         float res=1;
9         float expoente;
10    }
11    <write> "Insira a base: ";
12    <read> base;
13    <write> "Insira o expoente: ";
14    <read> expoente;
15    <if> expoente>=0 <do> {
16        <while> expoente>0 <do> {
17            res = res * base ;
18            expoente=expoente-1;
19        }
20    }
21    <else> {
22        <while> expoente<0 <do> {
23            res = res / base ;
24            expoente=expoente+1;
25        }
26    }
27    ;
28    <ret> "O resultado proveniente da potencia  o  : " + res;
29 }
30
31
32 ret = <call> potencia;
```

```
33
34 <write> ret;
```

C.1.2 Ficheiro .vm

Listing C.2: Resultado da compilação do programa em código Joma para calcular uma potência e indicar o tamanho do valor comparando-o com 50 e 100

```
1 pushs ""
2 start
3 jump af0
4 potencia :
5 pushf 0.0
6 pushi 1
7 itof
8 pushf 0.0
9 pushs "Insira a base: "
10 writes
11 read
12 dup 1
13 atof
14 storel 0
15 free
16 pushs "Insira o expoente: "
17 writes
18 read
19 dup 1
20 atof
21 storel 2
22 free
23 pushl 2
24 pushi 0
25 itof
26 supeq
27 jz e1
28 c2 :
29 pushl 2
30 pushi 0
31 itof
32 sup
33 jz e3
34 pushl 1
35 pushl 0
36 fmul
37 storel 1
38 pushl 2
39 pushi 1
40 itof
41 fsub
42 storel 2
43 jump c2
44 e3 :
45 jump t1
```

```
46 e1 :
47 c4 :
48 pushl 2
49 pushi 0
50 itof
51 inf
52 jz e5
53 pushl 1
54 pushl 0
55 fdiv
56 storel 1
57 pushi 1
58 itof
59 pushl 2
60 fadd
61 storel 2
62 jump c4
63 e5 :
64 t1 :
65 pushl 1
66 strf
67 pushs "O resultado proveniente da potencia  o  : "
68 concat
69 storel -1
70 return
71 af0 :
72 pushs ""
73 pusha potencia
74 call
75 storeg 0
76 pushg 0
77 writes
```

C.2 Ficheiro escrito em Joma com menu de utilização apresentando vários programas que o Utilizador pode testar como pedidos no enunciado

C.2.1 Ficheiro .jm

Listing C.3: Programa em código Joma

```
1 {
2     int read=1;
3     string output;
4 }
5
6 <func> readsquare {
7     {
8         float lado1;
9         float lado2;
10        float lado3;
11        float lado4;
```

```
12      }
13
14      <write> "Insira o comprimento de um lado:";
15      <read> lado1;
16
17      <write> "Insira o comprimento de um lado:";
18      <read> lado2;
19
20      <write> "Insira o comprimento de um lado:";
21      <read> lado3;
22
23      <write> "Insira o comprimento de um lado:";
24      <read> lado4;
25
26      <if> lado1==lado2 & lado2==lado3 & lado3==lado4 <do> {
27          <ret> "Podem ser lados de um quadrado\n";
28      }
29      <else> {
30          <ret> "N o podem ser lados de um quadrado\n";
31      }
32      ;
33  }
34
35  <func> readminimum {
36      {
37          int quantos_ler;
38          int i;
39          float atual;
40          float menor;
41      }
42      <write> "Quantos n meros deseja ler?";
43      <read> quantos_ler;
44      <for>(i=0;i<quantos_ler;i=i+1;){
45          <write> "Insira um n mero:";
46          <read> atual;
47          <if> atual<menor | i==0 <do> {
48              menor=atual;
49          }
50          ;
51      }
52      <ret> "O menor valor lido   :" + menor + "\n";
53  }
54
55  <func> calcprodutorio {
56      {
57          int how_many_to_read;
58          float produtorio=1;
59          float atual;
60          int i;
61      }
62      <write> "Quantos n meros deseja ler?";
63      <read> how_many_to_read;
64      <for>(i=0;i<how_many_to_read;i=i+1;){
65          <read>atual;
```

```
66          produtorio=produtorio*atual;
67      }
68      <ret> "O produt rio dos valores inseridos   : " + produtorio;
69  }
70
71  <func> readodd {
72      {
73          int quantos_ler;
74          int i;
75          int quantos;
76          string res;
77      }
78
79      <write> "Quantos n meros deseja ler?";
80      <read> quantos_ler;
81
82      <for>(i=1;i<=quantos_ler;i=i+1;){
83          <if> i%2==1 <do> {
84              quantos=quantos+1;
85              res = "O valor " + i + 1;
86              <write> res + "    impar !\n";
87          }
88          ;
89      }
90      res = "O n mero de valores impares encontrados   : " + quantos;
91      <ret> res + "\n";
92  }
93
94  #Funcionalidade que poderia ser melhorada com a adi a  de acesso atrav s de
        vari veis
95
96  <func> readarray {
97      {
98          float[5] lidos;
99      }
100     <read> lidos[0];
101     <read> lidos[1];
102     <read> lidos[2];
103     <read> lidos[3];
104     <read> lidos[4];
105     <write> lidos[4];
106     <write> lidos[3];
107     <write> lidos[2];
108     <write> lidos[1];
109     <write> lidos[0];
110     <ret> "Fun  o concluida";
111 }
112
113 <func> potencia {
114     {
115         float base;
116         float res=1;
117         float expoente;
118     }
```

```
119     <write> "Insira a base: ";
120     <read> base;
121     <write> "Insira o expoente: ";
122     <read> expoente;
123     <if> expoente>=0 <do> {
124         <while> expoente>0 <do> {
125             res = res * base ;
126             expoente=expoente-1;
127         }
128     }
129     <else> {
130         <while> expoente<0 <do> {
131             res = res / base ;
132             expoente=expoente+1;
133         }
134     }
135     ;
136     <ret> "O resultado proveniente da potencia   o   : " + res;
137 }
138
139 <while> read>0 <do> {
140     <write>"0 - Sair\n";
141     <write>"1 - Ler 4 n meros e testar se podem pertencer a um quadrado\n";
142     <write>"2 - Ler N n meros e determinar o menor deles\n";
143     <write>"3 - Ler N n meros e determinar o seu produt rio\n";
144     <write>"4 - Contar e imprimir os N primeiros impares\n";
145     <write>"5 - Ler 5 n meros para um array e imprimir o inverso dos mesmos\n";
146     <write>"6 - Calcular potencia   o com base N e expoente E\n";
147     <write>"Insira a op   o da fun   o que deseja testar: ";
148     <read> read;
149     <if> read==1 <do> {
150         output = <call> readsquare;
151     }
152     <else> {
153         <if> read==2 <do> {
154             output = <call> readminimum;
155         }
156         <else> {
157             <if> read==3 <do> {
158                 output = <call> calcprodutorio;
159             }
160             <else> {
161                 <if> read==4 <do> {
162                     output = <call> readodd;
163                 }
164                 <else> {
165                     <if> read==5 <do> {
166                         output = <call> readarray;
167                     }
168                     <else> {
169                         <if> read==6 <do> {
170                             output = <call> potencia;
171                         }
172                         <else> {
```

```
173                                     output="Obrigado por usar o nosso programa!\n";
174                                 }
175                                 ;
176                             }
177                             ;
178                         }
179                         ;
180                     }
181                     ;
182                 }
183                 ;
184             }
185             ;
186             <write> output;
187 }
```

C.2.2 Ficheiro .vm

Listing C.4: Resultado da compilação do programa em código Joma

```
1  pushi 1
2  pushs ""
3  start
4  jump af0
5  readsquare :
6  pushf 0.0
7  pushf 0.0
8  pushf 0.0
9  pushf 0.0
10 pushs "Insira o comprimento de um lado:"
11 writes
12 read
13 dup 1
14 atof
15 storel 0
16 free
17 pushs "Insira o comprimento de um lado:"
18 writes
19 read
20 dup 1
21 atof
22 storel 1
23 free
24 pushs "Insira o comprimento de um lado:"
25 writes
26 read
27 dup 1
28 atof
29 storel 2
30 free
31 pushs "Insira o comprimento de um lado:"
32 writes
33 read
```

```
34 dup 1
35 atof
36 storel 3
37 free
38 pushl 0
39 pushl 1
40 equal
41 pushl 1
42 pushl 2
43 equal
44 mul
45 pushl 2
46 pushl 3
47 equal
48 mul
49 jz e1
50 pushs "Podem ser lados de um quadrado\n"
51 storel -1
52 return
53 jump t1
54 e1 :
55 pushs "N o podem ser lados de um quadrado\n"
56 storel -1
57 return
58 t1 :
59 af0 :
60 jump af2
61 readminimum :
62 pushi 0
63 pushi 0
64 pushf 0.0
65 pushf 0.0
66 pushs "Quantos n meros deseja ler?"
67 writes
68 read
69 dup 1
70 atoi
71 storel 0
72 free
73 pushi 0
74 storel 1
75 f3 :
76 pushl 1
77 pushl 0
78 inf
79 jz ff4
80 pushs "Insira um n mero:"
81 writes
82 read
83 dup 1
84 atof
85 storel 2
86 free
87 pushl 2
```

```
88 pushl 3
89 inf
90 pushl 1
91 itof
92 pushi 0
93 itof
94 equal
95 add
96 jz e5
97 pushl 2
98 storel 3
99 e5 :
100 pushi 1
101 pushl 1
102 add
103 storel 1
104 jump f3
105 ff4 :
106 pushs "\n"
107 pushl 3
108 strf
109 pushs "O menor valor lido  : "
110 concat
111 concat
112 storel -1
113 return
114 af2 :
115 jump af6
116 calcprodutorio :
117 pushi 0
118 pushi 1
119 itof
120 pushf 0.0
121 pushi 0
122 pushs "Quantos n meros deseja ler?"
123 writes
124 read
125 dup 1
126 atoi
127 storel 0
128 free
129 pushi 0
130 storel 3
131 f7 :
132 pushl 3
133 pushl 0
134 inf
135 jz ff8
136 read
137 dup 1
138 atof
139 storel 2
140 free
141 pushl 1
```

```
142 pushl 2
143 fmul
144 storel 1
145 pushi 1
146 pushl 3
147 add
148 storel 3
149 jump f7
150 ff8 :
151 pushl 1
152 strf
153 pushs "O produt rio dos valores inseridos  : "
154 concat
155 storel -1
156 return
157 af6 :
158 jump af9
159 readodd :
160 pushi 0
161 pushi 0
162 pushi 0
163 pushs ""
164 pushs "Quantos n meros deseja ler?"
165 writes
166 read
167 dup 1
168 atoi
169 storel 0
170 free
171 pushi 1
172 storel 1
173 f10 :
174 pushl 1
175 pushl 0
176 infeq
177 jz ff11
178 pushl 1
179 pushi 2
180 mod
181 pushi 1
182 equal
183 jz e12
184 pushi 1
185 pushl 2
186 add
187 storel 2
188 pushi 1
189 stri
190 pushl 1
191 stri
192 pushs "O valor "
193 concat
194 concat
195 storel 3
```

```
196 pushs "␣  ␣impar␣!\n"
197 pushl 3
198 concat
199 writes
200 e12 :
201 pushi 1
202 pushl 1
203 add
204 storel 1
205 jump f10
206 ff11 :
207 pushl 2
208 stri
209 pushs "O␣n mero␣de␣valores␣impares␣encontrados␣  :␣"
210 concat
211 storel 3
212 pushs "\n"
213 pushl 3
214 concat
215 storel -1
216 return
217 af9 :
218 jump af13
219 readarray :
220 pushf 0.0
221 pushf 0.0
222 pushf 0.0
223 pushf 0.0
224 pushf 0.0
225 read
226 dup 1
227 atof
228 storel 0
229 free
230 read
231 dup 1
232 atof
233 storel 1
234 free
235 read
236 dup 1
237 atof
238 storel 2
239 free
240 read
241 dup 1
242 atof
243 storel 3
244 free
245 read
246 dup 1
247 atof
248 storel 4
249 free
```

```
250 pushl 4
251 writef
252 pushl 3
253 writef
254 pushl 2
255 writef
256 pushl 1
257 writef
258 pushl 0
259 writef
260 pushs "Fun   o concluida"
261 storel -1
262 return
263 af13 :
264 jump af14
265 potencia :
266 pushf 0.0
267 pushi 1
268 itof
269 pushf 0.0
270 pushs "Insira a base: "
271 writes
272 read
273 dup 1
274 atof
275 storel 0
276 free
277 pushs "Insira o expoente: "
278 writes
279 read
280 dup 1
281 atof
282 storel 2
283 free
284 pushl 2
285 pushi 0
286 itof
287 supeq
288 jz e15
289 c16 :
290 pushl 2
291 pushi 0
292 itof
293 sup
294 jz e17
295 pushl 1
296 pushl 0
297 fmul
298 storel 1
299 pushl 2
300 pushi 1
301 itof
302 fsub
303 storel 2
```

```
304 jump c16
305 e17 :
306 jump t15
307 e15 :
308 c18 :
309 pushl 2
310 pushi 0
311 itof
312 inf
313 jz e19
314 pushl 1
315 pushl 0
316 fdiv
317 storel 1
318 pushi 1
319 itof
320 pushl 2
321 fadd
322 storel 2
323 jump c18
324 e19 :
325 t15 :
326 pushl 1
327 strf
328 pushs "O resultado proveniente da potencia  o  : "
329 concat
330 storel -1
331 return
332 af14 :
333 c20 :
334 pushg 0
335 pushi 0
336 sup
337 jz e21
338 pushs "0 - Sair\n"
339 writes
340 pushs "1 - Ler 4 n meros e testar se podem pertencer a um quadrado\n"
341 writes
342 pushs "2 - Ler N n meros e determinar o menor deles\n"
343 writes
344 pushs "3 - Ler N n meros e determinar o seu produt rio\n"
345 writes
346 pushs "4 - Contar e imprimir os N primeiros impares\n"
347 writes
348 pushs "5 - Ler 5 n meros para um array e imprimir o inverso dos mesmos\n"
349 writes
350 pushs "6 - Calcular potencia  o com base N e expoente E\n"
351 writes
352 pushs "Insira a op  o da fun  o que deseja testar: "
353 writes
354 read
355 dup 1
356 atoi
357 storeg 0
```

```
358 free
359 pushg 0
360 pushi 1
361 equal
362 jz e22
363 pushs ""
364 pusha readsquare
365 call
366 storeg 1
367 jump t22
368 e22 :
369 pushg 0
370 pushi 2
371 equal
372 jz e23
373 pushs ""
374 pusha readminimum
375 call
376 storeg 1
377 jump t23
378 e23 :
379 pushg 0
380 pushi 3
381 equal
382 jz e24
383 pushs ""
384 pusha calcprodutorio
385 call
386 storeg 1
387 jump t24
388 e24 :
389 pushg 0
390 pushi 4
391 equal
392 jz e25
393 pushs ""
394 pusha readodd
395 call
396 storeg 1
397 jump t25
398 e25 :
399 pushg 0
400 pushi 5
401 equal
402 jz e26
403 pushs ""
404 pusha readarray
405 call
406 storeg 1
407 jump t26
408 e26 :
409 pushg 0
410 pushi 6
411 equal
```

```
412 jz e27
413 pushs ""
414 pusha potencia
415 call
416 storeg 1
417 jump t27
418 e27 :
419 pushs "Obrigado por usar o nosso programa!\n"
420 storeg 1
421 t27 :
422 t26 :
423 t25 :
424 t24 :
425 t23 :
426 t22 :
427 pushg 1
428 writes
429 jump c20
430 e21 :
```

Bibliografia

[1] Re library documentation. https://docs.python.org/3/library/re.html. Consultado: 2021-05-24.

[2] Pedro Rangel Henriques. Documentação da unidade curricular de pl. https://docs.google.com/document/d/1rvaZ2400C60EnWTJo3L7Lex8ONTxFW5Nh7WcGgiyORw/. Consultado: 2021-05-21.