



Escola de Engenharia  
**Universidade do Minho**

DEPARTAMENTO DE INFORMÁTICA  
**Mestrado Integrado em Engenharia  
Informática**  
*Processamento de Linguagens*

---

## TRABALHO PRÁTICO N<sup>o</sup> 2

*Implementação de compilador para uma linguagem de  
programação imperativa simples em YACC*

**Bruno Pereira**  
Aluno n<sup>o</sup> 72628

**Ricardo Oliveira**  
Aluno n<sup>o</sup> 58657

*Braga, 15 de Junho de 2016*

## **Resumo**

# Conteúdo

<b>Introdução</b>	<b>3</b>
<b>1 Análise do Problema</b>	<b>4</b>
1.1 Especificação dos requisitos	4
1.2 Pedidos	4
1.3 Dados	4
1.4 Relações	5
<b>2 Desenho e implementação da solução</b>	<b>7</b>
2.1 Desenho	7
2.1.1 Gramática Independente de Contexto	7
2.1.1.1 Axioma	7
2.1.1.2 Declarações de variáveis	8
2.1.1.3 Expressões	8
2.1.1.4 Instruções	9
2.1.2 Análise Semântica Estática	10
2.1.3 Análise Semântica Estática	10
2.1.4 Declarações	11
2.1.4.1 Fator	11
2.1.4.2 Termo	12
2.1.4.3 Expressão	13
2.1.4.4 Atribuição	13
2.1.4.5 Instruções	14
2.2 Implementação	15
2.2.1 Analisador léxico	15
2.2.1.1 Expressões Regulares	15
2.2.2 Analisador sintático e tradutor	17
2.2.2.1 Estruturas de dados	17
2.2.2.2 Algoritmos	19
<b>3 Testes e Resultados</b>	<b>28</b>
3.1 Resultados	28
3.1.1 Variáveis	28
3.1.2 Estruturas de Controlo	31
3.1.3 Estruturas de Controlo — Tipos	33
3.1.4 Expressões	35
3.2 Alternativas, Decisões e Problemas de Implementação	37
<b>Conclusão</b>	<b>38</b>

<b>Bibliografia</b>	<b>39</b>
<b>ANEXOS</b>	<b>41</b>
<b>A Gramática para linguagem criada</b>	<b>41</b>
<b>B Código do analisador léxico do <i>Flex</i></b>	<b>43</b>
<b>C Código do analisador sintático e tradutor do <i>YACC</i></b>	<b>45</b>
<b>D Código gerado a partir do tradutor do <i>YACC</i>, para os exemplos pedidos</b>	<b>60</b>
D.1 Maior de três números . . . . .	60
D.1.1 LPIS . . . . .	60
D.1.2 VM . . . . .	61
D.2 Somatório de N números . . . . .	61
D.2.1 LPIS . . . . .	61
D.2.2 VM . . . . .	62
D.3 Sequência de pares de N números dados . . . . .	63
D.3.1 LPIS . . . . .	63
D.3.2 VM . . . . .	64
D.4 Ordenação de <i>array</i> de tamanho N– <i>Insertion Sort</i> . . . . .	64
D.4.1 LPIS . . . . .	64
D.4.2 VM . . . . .	66
D.5 Média e máximo de uma matriz [N] [M] . . . . .	68
D.5.1 LPIS . . . . .	68
D.5.2 VM . . . . .	69

# Introdução

O presente relatório tem como objetivo documentar o processo de desenvolvimento de uma linguagem de programação imperativa simples, ou LPIS, e respectivo compilador, que deve ser capaz de gerar pseudo-código Assembly da máquina virtual VM, utilizada neste projeto. Para este fim, é necessário criar uma gramática independente de contexto que defina a linguagem, e estabelecer as regras de tradução para o Assembly da VM.

## Metas e objetivos

Este projeto pretende aumentar a experiência nos campos da engenharia de linguagens e programação generativa, através do desenvolvimento de linguagens e a utilização de geradores de compiladores baseados em gramáticas tradutores, o Yacc neste caso.

Adicionalmente, este projeto tem como objetivos secundários aumentar a capacidade de desenvolvimento de gramáticas independentes de contexto, e melhorar o uso do ambiente Linux e da linguagem imperativa C.

## Estrutura do Relatório

O relatório está dividido em três capítulos, correspondentes á análise do problema, á implementação da solução, e ao resultado dos testes efetuados, por esta ordem. No primeiro capítulo, *Análise do Problema*, são expostos os requisitos do problema apresentado, e discutidas as estratégias utilizadas para a consequente implementação da solução do problema. O capítulo dois, *Desenho e Implementação da Solução*, apresenta a gramática definida para a LPIS e explicita o funcionamento da análise léxica, sintática e semântica da linguagem criada. Este capítulo termina com a especificação das estruturas de dados e algoritmos usados na implementação final da solução. Por fim, o capítulo *Testes e Resultados* expõe o resultado dos testes requeridos á demonstração do funcionamento da linguagem e compilador desenvolvidos durante o projeto.

# Capítulo 1

## Análise do Problema

### 1.1 Especificação dos requisitos

O desafio deste projeto consiste na criação de uma linguagem de programação imperativa simples (LPIS) e respetivo compilador. Para tal é necessário criar uma gramática em *Bakus-Naur Form*–BNF–, definir símbolos terminais e não terminais, e desenvolver o analisador léxico, seguido do desenvolvimento do analisador sintático, com base nas regras da gramática, tendo, de igual modo, em conta a análise léxica. O compilador da linguagem irá incorporar ambas as análises supramencionadas, e procederá a uma análise de ações semânticas e á geração do código. O código gerado será pseudo-código da maquina virtual VM, o analisador léxico será elaborado no *Flex*, e o YACC será usado para a geração do código e análises sintática e semântica.

### 1.2 Pedidos

Para a linguagem alvo deste projeto, pede-se que, no mínimo, permita:

- Declaração e manuseamento de variáveis atômicas do tipo inteiro que permitam a realização de operações aritméticas, relacionais e lógicas;
- Declaração e manuseamento variáveis estruturadas do tipo *array* de inteiros, com 1 ou 2 dimensões, que permitam apenas operações de indexação;
- Realização de instruções algorítmicas básicas como a atribuição de expressões a variáveis;
- Leitura do *stdin* e escrita no *stdout*;
- Realização de instruções de controlo do fluxo de execução que permitam aninhamento;

Opcionalmente, a linguagem definida deve ser capaz de definir e invocar subprogramas sem parâmetros, mas que possam retornar um resultado atómico.

Pede-se também que, qualquer variável não pode ser redeclarada.

### 1.3 Dados

Uma linguagem imperativa completa deve de permitir pelo menos duas estruturas de controlo:

- Estruturas de fluxo condicional (*if then else*);

- Estruturas cíclicas (*while*);

Por uma questão de simplificação do código, poderá adicionalmente permitir a estrutura *if then* e a estrutura *do while*.

De igual modo, permitindo a linguagem acesso a estruturas do tipo *array*, é necessário ter em conta que qualquer *array*, independentemente da dimensão, é representado em memória como um único *array* de uma dimensão. Contudo, é necessário estabelecer regras para o acesso a arrays uni e bidimensionais. Para garantir a eficácia da linguagem, considerar-se-á que o acesso será feito em *row major* para arrays bidimensionais.

Genericamente, o acesso a um *array* pode ser representado por a fórmula  $A[i] = b + w * (i - lb)$ , sendo:

- $b$  o endereço base;
- $w$  o tamanho do elemento;
- $i$  o índice do elemento;
- $lb$  o limite inferior na memória;

No caso do acesso em questão,  $w = 1$  e  $lb = 0$ , logo  $A[i] = b + i$ .

Para um *array* bidimensional temos  $A[i][j] = b + w[N(i - lr) + (j - lc)]$ , onde:

- $b$  é o endereço base;
- $i$  é o índice da linha do elemento;
- $j$  é o índice da coluna do elemento;
- $w$  é o tamanho do elemento em bytes;
- $lr$  é o limite inferior da linha;
- $lc$  é o limite inferior da coluna;
- $N$  é o número máximo de linhas;

Assumindo  $w = 1$  e  $lr = lc = 0$ , temos  $A[i][j] = b + N * i + j$ .

## 1.4 Relações

Para o cálculo das expressões é necessário ter em conta algumas propriedades de cada tipo de expressão. Ou seja, é necessário verificar os tipos atômicos (variáveis, constantes, elementos de *arrays*), que assumimos como inteiros, e os resultados das expressões por inferência. Assim:

- Para as expressões aritméticas (soma, subtração, multiplicação, divisão inteira e módulo), bem como os elementos constituintes da expressão, o tipo deverá ser um inteiro;
- Para as expressões relacionais (maior, menor, maior ou igual, menor ou igual, igual e diferente), os elementos da expressão deverão ser do tipo inteiro e o resultado um valor booleano;
- Para as expressões lógicas, os elementos deverão ser booleanos e o resultado deverá também ser booleano;

De notar que várias expressões podem ser compostas, cuja verificação estará descrita na análise semântica. Adicionalmente, existe uma relação de precedência das operações, bem como de fatores. Um fator é uma expressão aninhada, um `//` ou uma variável. De igual modo, subprogramas e operações unárias? (uma vez que são funções) fazem parte deste conjunto. Consequentemente, um fator é prioritário em relação a todas as operações, uma vez que é o elemento atômico de uma expressão.

Em seguida, temos os termos, que são compostos por operações multiplicativas entre fatores, sendo estas a multiplicação, a divisão inteira, e módulo. Poderá também ser incluída a operação *E LÓGICO*, por razões que posteriormente serão explicadas. As expressões seguintes na escala de prioridades são as expressões aditivas (soma e subtração). Poderá ser incluída a operação *OU LÓGICO*, por razões que tal como a inclusão de *E LÓGICO*, que serão posteriormente explicitadas. As expressões de menor prioridade são as expressões relacionais (*menor*, *maior*, *menor ou igual*, *maior ou igual*, *igualdade* e *desigualdade*).

Para a execução do programa é necessário definir as instruções e operações que o definem. O programa pode efetuar cálculos utilizando as expressões previamente mencionadas. No entanto, é necessário guardar o resultado. Nas linguagem imperativas, a instrução base é a atribuição, que pode ser atribuição do valor de uma expressão a uma variável. Esta poderá ser uma variável atômica ou a posição de um *array*. As restantes instruções terão por base estes cálculos de expressões e atribuições, usando restrições de controlo de fluxo, tais como *if then else*, *while* e *do while*.

Devido à necessidade de armazenar o valor de expressões, torna-se necessário declarar as variáveis, alocando espaço em memória para as mesmas. As declarações podem designar o nome da variável, o seu tipo, e o seu valor, não haver redeclarações da mesma variável. Neste projeto, considerar-se-á que todas as variáveis são globais, ou seja, declaradas antes da execução do programa, e assumem número inteiro.

Para subprogramas, há que ter em conta que podem ter ou não parâmetros, e devolver ou não um valor. Visto que a alocação de memória é efetuada numa *stack* virtual, é necessário, no momento da sua invocação, criar uma *frame* no topo da *stack* com todas as variáveis locais declaradas, e parâmetros alocados em memória. Para identificar o programa principal, a linguagem usa um sistema de níveis, em que um nível assume o valor 0 para o programa principal ou o valor 1 para subprogramas. Por último, o início e fim de cada programa ou subprograma deve estar devidamente assinalado.



# Capítulo 2

## Desenho e implementação da solução

### 2.1 Desenho

#### 2.1.1 Gramática Independente de Contexto

O conjunto dos símbolos terminais da gramática é o que se segue:

```
T = { id,          num,      string
      BEGINNING, END,      VAR,
      NOT,         AND,     OR,
      READ,        WRITE,   IF,
      WHILE,       DO,
      ELSE,        '[',     ']',
      ';',         ',',     '(',
      ')',         '*',     '/',
      '\%',        '{',     '}',
      '+',         '-',     '<',
      '>',         '>=',   '<=',
      '==',        '!=',    '='
    }
```

O conjunto dos símbolos não-terminais da gramática é o que se segue:

```
NT = {Program,      Declarations, Body,
      InstructionsList, Declaration, DeclarationsList,
      Factor,        ExpAdditiv,  Exp,
      Variable,      Term,        Attribution,
      Instruction,   Else,        Constant
    }
```

##### 2.1.1.1 Axioma

Nesta linguagem, um programa é composto por declarações e um corpo.

$\langle \textit{Program} \rangle ::= \langle \textit{Declarations} \rangle \langle \textit{Body} \rangle$

O corpo do programa terá sempre que ter as palavras reservadas *BEGIN*, para iniciar a execução do programa, e *END*, para terminar a execução do programa. Entre estas duas palavras reservadas estará um conjunto de instruções.

$\langle \textit{Body} \rangle ::= \text{'BEGIN'} \langle \textit{InstructionList} \rangle \text{'END'}$

### 2.1.1.2 Declarações de variáveis

Assumiu-se que as variáveis seriam todas do tipo inteiro, tendo estas um identificador, podendo ser variáveis, *arrays* unidimensionais, ou *arrays* bidimensionais. O tamanho dos *arrays* será sempre um valor não negativo.

$$\begin{aligned}\langle Declaration \rangle &::= \langle id \rangle \\ &| \langle id \rangle \text{ ' [' } \langle num \rangle \text{ ' ] ' } \\ &| \langle id \rangle \text{ ' [' } \langle num \rangle \text{ ' ] ' ' [' } \langle num \rangle \text{ ' ] ' }\end{aligned}$$

Uma ou mais declarações formam um conjunto de declarações. Note-se que é mandatório pelo menos uma declaração.

$$\begin{aligned}\langle DeclarationsList \rangle &::= \langle Declaration \rangle \\ &| \langle DeclarationsList \rangle \text{ ' , ' } \langle Declaration \rangle\end{aligned}$$

As declarações devem começar sempre pela palavra reservada *VAR*.

$$\langle Declarations \rangle ::= \text{ 'VAR' } \langle DeclarationsList \rangle \text{ ' ; ' }$$

### 2.1.1.3 Expressões

Uma constante é um número não negativo.

$$\langle Constant \rangle ::= \langle num \rangle$$

Uma variável será sempre um identificador, um *array*, com uma expressão inteira no seu índice, ou índices, se for multidimensional.

$$\begin{aligned}\langle Variable \rangle &::= \langle id \rangle \\ &| \langle id \rangle \text{ ' [' } \langle ExpAdditiv \rangle \text{ ' ] ' } \\ &| \langle id \rangle \text{ ' [' } \langle ExpAdditiv \rangle \text{ ' ] ' ' [' } \langle ExpAdditiv \rangle \text{ ' ] ' }\end{aligned}$$

Um fator pode ser uma constante, uma variável, uma expressão, uma expressão negativa, ou a negação de uma expressão.

$$\begin{aligned}\langle Factor \rangle &::= \langle Constant \rangle \\ &| \langle Variable \rangle \\ &| \text{ ' ( ' } \langle Exp \rangle \text{ ' ) ' } \\ &| \text{ ' ( ' ' - ' } \langle Exp \rangle \text{ ' ) ' } \\ &| \text{ 'NOT' } \langle Exp \rangle\end{aligned}$$

Um termo será sempre um conjunto de um ou mais fatores, em que as operações que o compõem serão sempre multiplicativas. Note-se que dado não haver instruções lógicas na VM, o *AND* terá que ser uma multiplicação entre valores inteiros entre 0 e 1.

$$\begin{aligned}\langle Term \rangle &::= \langle Factor \rangle \\ &| \langle Term \rangle \text{ ' * ' } \langle Factor \rangle \\ &| \langle Term \rangle \text{ ' / ' } \langle Factor \rangle \\ &| \langle Term \rangle \text{ ' \% ' } \langle Factor \rangle \\ &| \langle Term \rangle \text{ 'AND' } \langle Factor \rangle\end{aligned}$$

Uma expressão aditiva será sempre um conjunto de um ou mais termos, em que as operações que o compõem serão sempre aditivas. Note-se que dado não haver instruções lógicas na VM, o *OR* terá que ser uma soma entre valores inteiros não negativos.

$$\begin{aligned} \langle \text{ExpAdditiv} \rangle &::= \langle \text{Term} \rangle \\ &| \langle \text{ExpAdditiv} \rangle \text{ '+' } \langle \text{Term} \rangle \\ &| \langle \text{ExpAdditiv} \rangle \text{ '-' } \langle \text{Term} \rangle \\ &| \langle \text{ExpAdditiv} \rangle \text{ 'OR' } \langle \text{Term} \rangle \end{aligned}$$

Uma expressão será uma expressão aditiva ou duas expressões aditivas com determinada relação.

$$\begin{aligned} \langle \text{Exp} \rangle &::= \langle \text{ExpAdditiv} \rangle \\ &| \langle \text{ExpAdditiv} \rangle \text{ '>' } \langle \text{ExpAdditiv} \rangle \\ &| \langle \text{ExpAdditiv} \rangle \text{ '<' } \langle \text{ExpAdditiv} \rangle \\ &| \langle \text{ExpAdditiv} \rangle \text{ '>=' } \langle \text{ExpAdditiv} \rangle \\ &| \langle \text{ExpAdditiv} \rangle \text{ '<=' } \langle \text{ExpAdditiv} \rangle \\ &| \langle \text{ExpAdditiv} \rangle \text{ '==' } \langle \text{ExpAdditiv} \rangle \\ &| \langle \text{ExpAdditiv} \rangle \text{ '!=' } \langle \text{ExpAdditiv} \rangle \end{aligned}$$

#### 2.1.1.4 Instruções

Uma atribuição será sempre uma variável a tomar o valor de uma expressão inteira.

$$\langle \text{Attribution} \rangle ::= \langle \text{Variable} \rangle \text{ '=' } \langle \text{ExpAdditiv} \rangle$$

Uma instrução pode ser uma atribuição, a leitura de uma variável de *stdin*, a escrita de um valor inteiro ou uma *string* no *stdout*, ou uma estrutura de controlo com a avaliação de uma expressão booleana, com um conjunto de instruções associados a cada condição.

$$\begin{aligned} \langle \text{Instruction} \rangle &::= \langle \text{Attribution} \rangle \text{ ';' } \\ &| \text{'READ' } \langle \text{Variable} \rangle \text{ ';' } \\ &| \text{'WRITE' } \langle \text{ExpAdditiv} \rangle \text{ ';' } \\ &| \text{'WRITE' } \langle \text{string} \rangle \text{ ';' } \\ &| \text{'IF' ' (' } \langle \text{Exp} \rangle \text{ ') ' '{' } \langle \text{InstructionsList} \rangle \text{ '}' } \langle \text{Else} \rangle \\ &| \text{'WHILE' ' (' } \langle \text{Exp} \rangle \text{ ') ' '{' } \langle \text{InstructionsList} \rangle \text{ '}' } \\ &| \text{'DO' '{' } \langle \text{InstructionsList} \rangle \text{ '}' 'WHILE' ' (' } \langle \text{Exp} \rangle \text{ ') ' ';' } \end{aligned}$$

A bloco seguinte representa a existência ou não de instruções alternativas a um *if*.

$$\begin{aligned} \langle \text{Else} \rangle &::= \langle \rangle \\ &| \text{'ELSE' '' } \langle \text{InstructionsList} \rangle \text{ ''} \end{aligned}$$

Uma lista de instruções é um conjunto de uma ou mais instruções. Note-se que a existência de instruções é mandatória, isto é, um programa ou uma estrutura de controlo terá sempre pelo menos uma instrução.

$$\begin{aligned} \langle \text{InstructionList} \rangle &::= \langle \text{Instruction} \rangle \\ &| \langle \text{InstructionList} \rangle \langle \text{Instruction} \rangle \end{aligned}$$

### 2.1.2 Análise Semântica Estática

A análise semântica estática complementa a análise lexical e sintática, pois existem situações em que a apesar de a análise léxica e sintática estarem corretas, as sequências de símbolos não têm sentido. A título de exemplo, a data 2016-45-00 está sintaticamente correta pois segue o formato `aaaa-mm-dd`, mas não tem sentido do ponto de vista semântico.

Na *LPIS* existem situações em que a gramática e análise léxica não são suficientes, nomeadamente na verificação de tipos, ou seja, se estes são consistentes na sua definição. Anteriormente, é mencionada a inferência de tipos em fatores, variáveis, arrays, aditivos, termos, expressões e expressões relacionais. Assim é necessário não só verificar os elementos de cada operação binária, como inferir o tipo do seu resultado. Como já foi mencionado, esta *LPIS* apenas permite valores inteiros, mas no entanto o resultado das expressões lógicas e relacionais são do tipo booleano. Por uma questão de consistência, considera-se que as instruções terão o tipo *Any*.

Assim, as instruções assumem o tipo *Any*, variáveis atômicas e arrays o tipo *Integer* (exceto subprogramas sem valor de retorno), e o resultado de uma operação aditiva entre fatores como variáveis e arrays assumirá o tipo *Integer*, bem como o tipo de cada de cada membro da operação binária aditiva. O resultado de uma operação multiplicativa entre termos (resultantes da operação aditiva) também tomará o tipo *Integer*, bem como ambos os membros da operação binária em questão. Seguidamente, o resultado de expressões relacionais deverá ser do tipo *Boolean*. No entanto, os membros desta operação binária assumirão o tipo *Integer*. Para concluir, o resultado de uma expressão lógica assumirá o tipo *Boolean* e ambos os membros da operação binária assumem também o tipo *Boolean*.

Para além da verificação de tipos, a análise semântica deve assegurar a existência de etiquetas, ou *labels*, de referência no resultado da geração de código. Dado que a linguagem em causa tem quatro tipos de estruturas de controlo de fluxo, existirá um mecanismo que cria as *labels* para cada tipo de estrutura de controlo e que permita o aninhamento das mesmas. Ou seja, as etiquetas devem ter a referência do nível em que estão, e devem ser colocadas no seu devido lugar. Posteriormente, será descrito o algoritmo e a implementação do mecanismo de criação de *labels*.

Adicionalmente, as estruturas de controlo devem ser verificadas para confirmar a sua devida utilização. Por exemplo, se existem *breaks* fora de um *loop* ou *switch*.

Finalmente, é necessário considerar outro tipo de análise semântica: a análise semântica dinâmica. Ao contrário da anterior, este tipo de análise não é efetuada em tempo de compilação, mas sim em tempo de execução. Existem exemplos retirados da gramática da linguagem alvo deste projeto, que não serão considerados, visto que a máquina virtual VM já os inclui, como é o caso da divisão por zero, em que os valores retirados da pilha não têm o tipo esperado, ou acessos indefinidos a uma ? de código. Na especificação léxica da linguagem, definiu-se que todos os números tomarão valores não negativos, e previu-se a existência de números negativos na gramática a uma operação unária?. Deste modo não é necessária a especificação semântica estática de declarações de arrays com tamanho zero, resultados negativos em cálculos de índices e divisões por zero, uma vez que estes casos fazem parte da análise semântica dinâmica.

### 2.1.3 Análise Semântica Estática

Program : Declarations Body

Nesta produção é imprimido no *stdout*, o resultado dos blocos que vêm de baixo da árvore sintática, através de *Body*.

Body : BEGINNING

Esta produção tem uma regra intermédia, em que imprime no *stdout* a *string* 'start'. A ação é para ser executada de imediato, aquando do reconhecimento da palavra reservada.

InstructionsList END

O restante da produção, junta tudo dos blocos construídos a partir de baixo da árvore sintática, concatenando a *string* ‘stop’ no fim. O resultado vai para cima da árvore sintática.

## 2.1.4 Declarações

**Declaration : id** Nesta produção é verificado se o identificador não existe. Se não existe o identificador é adicionado à tabela de identificadores, com o o tipo, classe (neste caso *Variable*) e com o nível *Program*. A *string* ‘pushi 0’ é impressa no *stdout*.

| id '[' num ']' Nesta produção é verificado se o identificador não existe. Se não existe o identificador é adicionado à tabela de identificadores, com o o tipo, classe (neste caso *Array*) e com o nível *Program* e com o seu tamanho. A *string* ‘pushn’ concatenada com o tamanho é impressa no *stdout*.

| id '[' num ']' '[' num ']' Nesta produção é verificado se o identificador não existe. Se não existe o identificador é adicionado à tabela de identificadores, com o o tipo, classe (neste caso *Matrix*) e com o nível *Program* e com o seu tamanho, e o número de linhas. A *string* ‘pushn’ concatenada com o tamanho é impressa no *stdout*.

**Declarations : VAR DeclarationsList ‘;’**

Nesta produção o valor do bloco construído de baixo da árvore é passado para cima.

**DeclarationsList : Declaration**

Nesta produção o valor do bloco construído de baixo da árvore é passado para cima.

| DeclarationsList ‘,’ Declaration

Nesta produção o valor do bloco construído de baixo da árvore é passado para cima, após juntar o que vem do lado da mão direita com o que vem do lado da mão esquerda.

**Variable : id**

Nesta produção é verificado se o identificador existe. Se existe o endereço é obtido da tabela de identificadores. A *string* ‘pushg’ concatenada com o endereço é impressa é juntada ao bloco. O tipo inteiro é passado para o nível acima da árvore sintática. Caso contrário, é lançado um erro.

| id '[' ExpAdditiv ']'

Nesta produção é verificado se o identificador existe. Se existe o endereço é obtido da tabela de identificadores. A *string* ‘pushgp’ concatenada com o endereço, mais a *string* ‘padd’ é junta ao bloco.. Caso contrário, é lançado um erro.

| id '[' ExpAdditiv ']' '[' ExpAdditiv ']'

Nesta produção é verificado se o identificador existe. Se existe o endereço é obtido da tabela de identificadores. A *string* ‘pushgp’ concatenada com o endereço, mais as *strings* ‘padd’, onde o que vem das expressões é adicionado ao bloco, seguido de ‘pushi’ com o número de linhas seguido de ‘mul’ e ‘add’ seguindo o algoritmo de calculo de endereços em *row major*. O tipo inteiro é passado para o nível acima da árvore sintática, juntamente com o apontador para o valor da entrada da tabela de identificadores. Caso contrário, é lançado um erro.

**Constant : num**

Nesta produção é criada a *string* ‘pushi’ concatenada com o valor numérico para cima na árvore sintática. O tipo também é passado para cima, como um tipo inteiro.

### 2.1.4.1 Fator

**Factor : Constant**

Nesta produção os valores das instruções geradas são passadas para *Factor*, herdando assim as propriedades de uma constante (tipo e instruções geradas).

| Variable

Nesta produção, tal como nas constantes os valores são passados para *Variable*. Note-se que o tipo de *Variable* é um triplo, ou seja passa as instruções geradas, o tipo e um apontador para a entrada resolvida no nível abaixo da árvore sintática.

| '(' '-' Exp )'

Para a operação unária negativa de números inteiros, é necessário verificar se o valor da expressão é do tipo inteiro. Caso contrário, há um erro. A ação semântica passa por obter da árvore sintática, todas as instruções geradas, e depois gerar a *string* 'pushi -1sub'. O tipo da operação é passado para o nível superior da árvore sintática.

| '(' Exp )'

Nesta produção, tanto os valores de tipo e instruções são passadas de baixo para cima da árvore sintática.

| NOT '(' Exp )'

Para a operação unária negativa de números inteiros, é necessário verificar se o valor da expressão é do tipo booleano. Caso contrário, há um erro. A ação semântica passa por obter da árvore sintática, todas as instruções geradas, e depois gerar a *string* 'not'. O tipo da operação é passado para o nível superior da árvore sintática.

#### 2.1.4.2 Termo

Term : Factor

Á semelhança de outras produções, os valores do tipo e instruções são passados de baixo para cima da árvore sintática. Portanto um termo herda as propriedades de um fator.

| Term '\*' Factor

Nesta produção, em ambos os membros da operação de multiplicação, os tipos são verificados, ou seja, se ambos são do tipo inteiro. Caso contrário, há um erro. As instruções que vem de ambos os membros geradas, logo antes de se gerar a *string* 'mul', juntando esta ao restante bloco de instruções. O tipo que é passado para o nível acima é um inteiro.

| Term '/' Factor

Nesta produção, em ambos os membros da operação de multiplicação, os tipos são verificados, ou seja, se ambos são do tipo inteiro. Caso contrário, há um erro. As instruções que vem de ambos os membros geradas, logo antes de se gerar a *string* 'div', juntando esta ao restante bloco de instruções. O tipo que é passado para o nível acima é um inteiro.

| Term '

Nesta produção, em ambos os membros da operação de multiplicação, os tipos são verificados, ou seja, se ambos são do tipo inteiro. Caso contrário, há um erro. As instruções que vem de ambos os membros geradas, logo antes de se gerar a *string* 'mod', juntando esta ao restante bloco de instruções. O tipo que é passado para o nível acima é um inteiro. | Term AND Factor

Nesta produção, em ambos os membros da operação de multiplicação, os tipos são verificados, ou seja, se ambos são do tipo inteiro. Caso contrário, há um erro. As instruções que vem de ambos os membros geradas, logo antes de se gerar a *string* 'mul', juntando esta ao restante bloco de instruções. O tipo que é passado para o nível acima é um booleano. -> EXP ADITIVA

ExpAdditiv : Term

Nesta produção os valores das instruções geradas são passadas para *ExpAdditiv*, herdando assim as propriedades de uma constante (tipo e instruções geradas).

| ExpAdditiv '+' Term

Nesta produção, em ambos os membros da operação de multiplicação, os tipos são verificados, ou seja, se ambos são do tipo inteiro. Caso contrário, há um erro. As instruções que vem de ambos os membros geradas, logo antes de se gerar a *string* 'add', juntando esta ao restante bloco de instruções. O tipo que é passado para o nível acima é um inteiro.

| ExpAdditiv '-' Term

Nesta produção, em ambos os membros da operação de multiplicação, os tipos são verificados, ou seja, se ambos são do tipo inteiro. Caso contrário, há um erro. As instruções que vem de ambos os membros geradas, logo antes de se gerar a *string* 'sub', juntando esta ao restante bloco de instruções. O tipo que é passado para o nível acima é um inteiro..

| ExpAdditiv OR Term

Nesta produção, em ambos os membros da operação de multiplicação, os tipos são verificados, ou seja, se ambos são do tipo inteiro. Caso contrário, há um erro. As instruções que vem de ambos os membros geradas, logo antes de se gerar a *string* 'add', juntando esta ao restante bloco de instruções. O tipo que é passado para o nível acima é um booleano.

### 2.1.4.3 Expressão

Exp : ExpAdditiv

Nesta produção, tanto os valores de tipo e instruções são passadas de baixo para cima da árvore sintática.

| ExpAdditiv L ExpAdditiv

Nesta produção, em ambos os membros da operação de multiplicação, os tipos são verificados, ou seja, se ambos são do tipo inteiro. Caso contrário, há um erro. As instruções que vem de ambos os membros geradas, logo antes de se gerar a *string* 'inf', juntando esta ao restante bloco de instruções. O tipo que é passado para o nível acima é um booleano.

| ExpAdditiv G ExpAdditiv Nesta produção, em ambos os membros da operação de multiplicação, os tipos são verificados, ou seja, se ambos são do tipo inteiro. Caso contrário, há um erro. As instruções que vem de ambos os membros geradas, logo antes de se gerar a *string* 'sup', juntando esta ao restante bloco de instruções. O tipo que é passado para o nível acima é um booleano.

| ExpAdditiv GEQ ExpAdditiv Nesta produção, em ambos os membros da operação de multiplicação, os tipos são verificados, ou seja, se ambos são do tipo inteiro. Caso contrário, há um erro. As instruções que vem de ambos os membros geradas, logo antes de se gerar a *string* 'supeq', juntando esta ao restante bloco de instruções. O tipo que é passado para o nível acima é um booleano.

| ExpAdditiv LEQ ExpAdditiv Nesta produção, em ambos os membros da operação de multiplicação, os tipos são verificados, ou seja, se ambos são do tipo inteiro. Caso contrário, há um erro. As instruções que vem de ambos os membros geradas, logo antes de se gerar a *string* 'infeq', juntando esta ao restante bloco de instruções. O tipo que é passado para o nível acima é um booleano.

| ExpAdditiv EQ ExpAdditiv Nesta produção, em ambos os membros da operação de multiplicação, os tipos são verificados, ou seja, se ambos são do tipo inteiro. Caso contrário, há um erro. As instruções que vem de ambos os membros geradas, logo antes de se gerar a *string* 'equal', juntando esta ao restante bloco de instruções. O tipo que é passado para o nível acima é um booleano.

| ExpAdditiv NEQ ExpAdditiv Nesta produção, em ambos os membros da operação de multiplicação, os tipos são verificados, ou seja, se ambos são do tipo inteiro. Caso contrário, há um erro. As instruções que vem de ambos os membros geradas, logo antes de se gerar as *string* com as *strings* 'equal' e 'not' concatenadas, juntando esta ao restante bloco de instruções. O tipo que é passado para o nível acima é um booleano.

### 2.1.4.4 Atribuição

Atribution : Variable '=' ExpAdditiv

Nesta produção são verificados os tipos, tanto da expressão inteira como da variável. Se forem ambos do tipo inteiro executam a ação, caso contrário gera-se uma erro. Após a verificação é verificada a categoria da variável, que vem na informação da entrada por apontador, vinda de baixo da

árvore sintática. Se a categoria for um *array* ou uma matriz, junta ao bloco a *string* ‘storen’. Caso contrário obtém o endereço do valor da entrada e junta ao bloco a *string* ‘storen’ concatenada com o endereço da variável da *stack*.

#### 2.1.4.5 Instruções

InstructionsList : Instruction

Nesta produção, o bloco da instrução é passado para cima da árvore sintática.

| InstructionsList Instruction

Nesta produção, o bloco da instrução é construído com base do que vem do lado da mão direita, com o lado da mão esquerda, passando para cima da árvore sintática, o resultado.

Instruction : Attribution ‘;’ Nesta produção, o bloco da atribuição é passado para cima da árvore sintática.

| READ Variable ‘;’

Nesta produção, é obtido de *Variable*, o endereço da *stack* de uma variável. Após obter de cima da árvore sintática, as instruções geradas a *string* ‘pushg %d read’ com o endereço da *stack* virtual é adicionada ao bloco.

| WRITE ExpAdditiv ‘;’

Nesta produção é verificado o tipo da expressão inteira. Caso for do tipo inteiro, executa a ação. Caso contrário, lança um erro. Após obter de cima da árvore sintática o bloco de intruções anteriores, concatena a este a *string* ‘writei’.

| WRITE string ‘;’

A ação nesta produção, é concatenar a *string* ‘writes’ ao bloco.

| IF ‘(’ Exp ‘)’

Esta produção tem uma regra intermédia. O tipo da expressão é avaliado se é booleano. Caso contrário é lançado um erro. A ação é juntar o que vem de *Exp*, juntar ao que vem de cima, e depois criar a *label* ‘jz l1level’ concatenada com a *string* que vem da intrução *add\_label*.

” InstructionsList ” Else

No final desta produção, o que vem da *InstructionList* e que vem do *Else* é passado para cima da árvore sintática.

| WHILE ‘(’ Exp ‘)’

Esta produção tem uma regra intermédia. O tipo da expressão é avaliado se é booleano. Caso contrário é lançado um erro. A ação é imprimir a *string* ‘whileloop’ conancatenado com o resultado da função *add\_label*, juntar o que vem de *Exp*, juntar ao que vem de cima, e depois criar a *label* ‘jz whiledone’ concatenada com a *string* que vem da instrução *get\_label*.

” InstructionsList ” No final desta produção, obtido o valor atual da *stack* de contadores, guardado numa variável temporária, sendo removida o *label* da *string*. Depois é junto o que o vem de *InstructionsList*, mais o valor guardado da variável temporária, concatenado com *jump whileloop* e o *whileloop*. Em seguida é removido um nível da *stack*

| DO ” InstructionsList ” WHILE ‘(’ Exp ‘)’ ‘;’

Else :

| ELSE ” InstructionsList ”



## 2.2 Implementação

### 2.2.1 Analisador léxico

Na fase de construir o analisador léxico, tomou-se em conta a análise anterior da gramática e definiu-se para cada símbolo terminal uma expressão regular, e respetiva ação associada.

#### 2.2.1.1 Expressões Regulares

Em primeira instância definiu-se o que era uma letra, um dígito e, tudo o que será para ser rejeitado da seguinte forma:

---

```
1  letra  [A-Za-z]
2  digito [0\ -9]
3  lixo  \.|\ \n
```

---

Em seguida definiram-se as expressões para captura dos símbolos terminais, para serem usadas como *tokens* no ficheiro *YACC*, definiram-se da seguinte forma:

- Captura de operadores de apenas um caractere;

---

```
1  [-/; , \[ \] + \ ( \) \{ \} \% =]
```

---

Com esta expressão regular são capturados os símbolos referentes às operações aditivas e multiplicativas, bem como caracteres usados para delimitar expressões e na construção de estruturas de controlo (chavetas e parêntesis). De igual modo, são capturados o símbolo unário de um número negativo, delimitadores de final de instruções (ponto e vírgula), separadores de declarações (vírgula), elementos de declaração de *arrays* (parêntesis retos) e o sinal de atribuição, que nesta linguagem é o sinal de igual.

A ação para esta expressão regular é devolver o valor do código ASCII-estendido de cada caractere capturado.

- Captura de operadores de operações lógicas e relacionais.

---

•	1	(OR)	{return OR; }
	2	(<)	{return L; }
	3	(>)	{return G; }
	4	(<=)	{return LEQ; }
	5	(>=)	{return GEQ; }
	6	(==)	{return EQ; }
	7	(!=)	{return NEQ; }

---

- Captura de palavra reserva para início e fim de programa.

---

1	(BEGINNING)	{return BEGINNING; }
2	(END)	{return END; }

---

A ação será retornar um valor numérico, que será atribuído aquando o *linking* do ficheiro *Flex* e do ficheiro *YACC*, no `yy.tab.c`.

- Captura das palavras reservadas para funções de leitura e escrita.

---

1	(READ)	{return READ; }
2	(WRITE)	{return WRITE; }

---

Ação a mesma que a anterior.

- Captura das palavras reservadas usadas para declarações e estruturas de controlo.

---

1	(VAR)	{return VAR; }
2	(WHILE)	{return WHILE; }
3	(IF)	{return IF; }
4	(ELSE)	{return ELSE; }
5	(DO)	{return DO; }

---

Ação: *ibidem*.

- Captura de uma *string*

`\ "[^"]+\\"`

Neste caso é atribuído ao valor da variável global `yyval` no elemento da união `char * val_string` uma cópia da *string*, sendo retornado um valor que será atribuído pelo *YACC*.

- Captura de um dígito

---

1 {digito}+

---

A ação é atribuir ao valor da variável global `yyval` no elemento da união `int val_nro` o valor do dígito capturado., sendo retornado um valor que será atribuído pelo *YACC*.

- Captura de um identificador.

---

1 {letra}+

---

Neste caso é atribuído ao valor da variável global `yyval` no elemento da união `char * val_id` uma cópia da palavra capturada, sendo retornado um valor que será atribuído pelo *YACC*.

- Outros

---

```
1 {lixo}                                {;}
```

---

Ação: ignorar.

## 2.2.2 Analisador sintático e tradutor

Para o analisador sintático, criaram-se algumas estruturas de suporte ao *parser*, nomeadamente uma *hashtable* para a tabela de identificadores, uma biblioteca para os dados referentes a cada identificador, uma biblioteca para suporte ao *parser*, com toda a informação relativa ao estado do *parser*—apontador de endereços na *stack* virtual, *stacks* para calculo do nível das *etiquetas*— e, adicionalmente definiram-se tipos enumerados para serem usados em transversalmente na aplicação.

### 2.2.2.1 Estruturas de dados

A biblioteca *entry* possui uma estrutura composta pelos campos mencionados em secções anteriores: tipo, classe, nível, que são tipos enumerados, e, endereço base, número de linhas máximo (caso seja uma matriz ) e tamanho máximo (para *arrays* unidimensionais e bidimensionais). Adicionalmente, considerou-se a criação de uma lista de argumentos, no entanto, por razões que serão posteriormente explicitadas, decidiu-se não se incluir.

As funções referentes à esta biblioteca, inicializam e desalocam memória e vão buscar os dados da estrutura, ou atualizam os dados desta estrutura. Para facilitar a criação de entradas na tabela, com diferentes tipos de classes (*array*, *matriz* e *variável*), criaram-se métodos que providenciam a criação da entrada por classe.

A biblioteca *program\_status* que guarda informação sobre o *parsing* tem o formato que se segue;

```
typedef struct stat
{
    char label                [MAX_CONDITION_ROW] [ MAX_LABEL ];
    int  label_stack          [MAX_CONDITION_ROW] [ MAX_LABEL_STACK ];
    int  label_number_size    [MAX_CONDITION_ROW] [ MAX_LABEL_STACK ];
    int  spointer             [MAX_CONDITION_ROW] [1];
    int  strpointer           [MAX_CONDITION_ROW] [1];
    int  size_label_string    [MAX_CONDITION_ROW] [1];
    int  addresspointer;
} Program_status;
```

Basicamente, a estrutura possui *arrays* bidimensionais para a representação das *stacks* das *labels*. Estas têm 4 linhas, ou seja, uma linha para cada tipo de estrutura de controlo. Deste modo, a variável *label* irá guardar a concatenação dos valores dos níveis de aninhamento, a variável *label\_stack* possuirá os contadores para cada nível de aninhamento e a variável *label\_number\_size* irá guardar o tamanho da da *string* resultante do valor de cada nível concatenado. A biblioteca possui funções comuns às *stacks*, como *pop*, *push*, *top* e, adicionalmente possui funções específicas para a manipulação da informação das *stacks*, que usa o tipo enumerado *CompoundInstruction* para aceder por índice às *stacks*.

As funções específicas para o cálculo das etiquetas são:

- `reset_label_stack` Esta função tem por âmbito reinicializar o contador após sair de uma ação semântica, como se poderá ver na secção sobre algoritmos, na posição em `stack[stack_pointer]`. Note-se que o `top` é em `stack[stack_pointer-1]`

- `increment_top_label_stack` Esta função incrementa o valor do contador na posição `stack_pointer - 1` da *stack* de contadores, ou seja, incrementa uma nova ocorrência no mesmo nível.

- `char *get_label`

Esta função obtém a *string* criada até ao momento, com os valores das ocorrências dos níveis concatenados na *string* da *label*.

- `char *push_label`

A função `pushlabel` incrementa o valor do `top` da *stack* de contadores, converte o valor numérico do `top` para uma *string*, calcula o tamanho desta *string* e guarda na *stack* de tamanhos das *strings*. Em seguida, concatena a *string* convertida ‘a *string* em construção, sendo esta copiada para ser retornada pela função. Adicionalmente, guarda o tamanho da *string* convertida na devida *stack* e avança o apontador da *string* em construção por esse tamanho.

- `pop_label`

Obtém o tamanho da *string* concatenada, guardada anteriormente na *stack* `label_number_size` e coloca caracteres nulos na pilha com a *string* para as *labels*, decrementando o apontador da pinha com a *string* no valor desse tamanho.

Note-se que, a descrição destas funções é fundamental para a compreensão do código, uma vez que serão utilizadas mais adiante no documento.

De igual modo, possui funções para o cálculo de endereços para cada classe de variável, onde o apontador para a *stack* virtual é incrementado pelo tamanho da variável. Outras funções são a inserção de um identificador com os seus devidos valores na tabela de identificadores, bem como a procura de um identificador e remoção de todos os identificadores. Por último, a função `check_type` compara dois tipos.

Além destas bibliotecas, foram definidos novos tipos, enumerados, no cabeçalho `types.h`, que são os seguintes:

- `CompoundInstruction`; Este tipo caracteriza tipos de estruturas de controlo diferentes para acesso por índice aos *arrays* multidimensionais, representam as diferentes *stacks* para as *labels*.

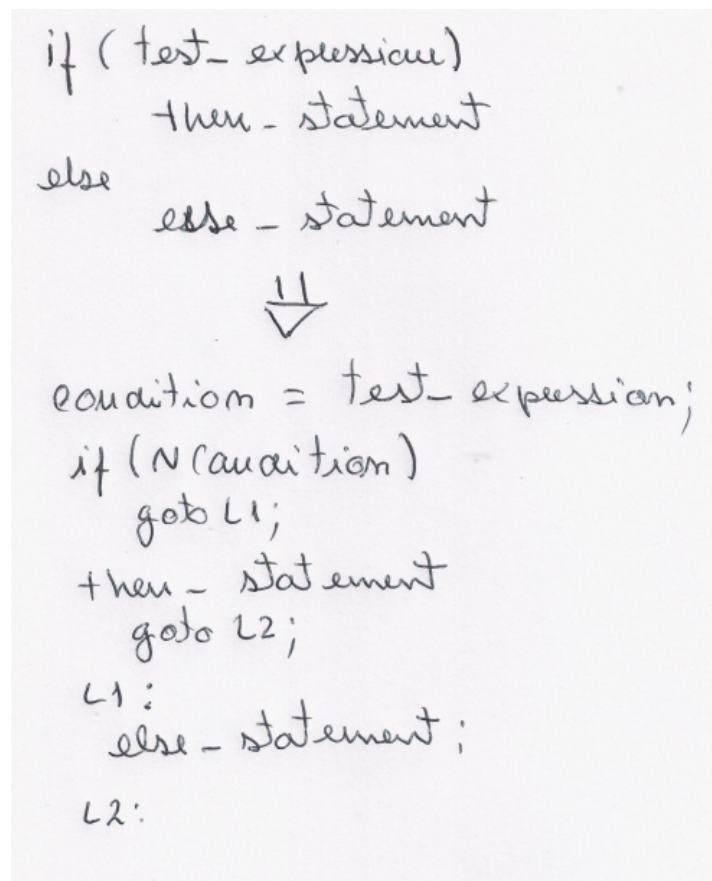
- `if_inst`
- `else_inst`
- `while_inst`
- `do_while_inst`

- `Class`;a O tipo `Class` serve para diferenciar as categorias, ou classes, de objetos que podem ser declarados e invocados.

- `Variable`
- `Array`
- `Matrix`

- Function
- Procedure
- Nothing
- Level; O nível tem o propósito de diferenciar programas e subprogramas.
  - Program
  - Subprogram
- Type; Os tipos das variáveis e expressões podem ser os seguintes:
  - Any
  - Integer
  - Boolean Embora se possa atribuir o tipo Any a instruções, não existe relevância para o fazer.

#### 2.2.2.2 Algoritmos



```

if ( test-expression )
    then - statement
else
    else - statement
    ↓
condition = test-expression;
if ( condition )
    goto L1;
then - statement
goto L2;
L1:
else - statement;
L2:
  
```

**Figura 2.1:** Algoritmo para if else

```

if (test-expression)
    then-statement
    ⇓
condition = test-expression;
if (¬ condition)
    goto L1;
then-statement;
L1;

```

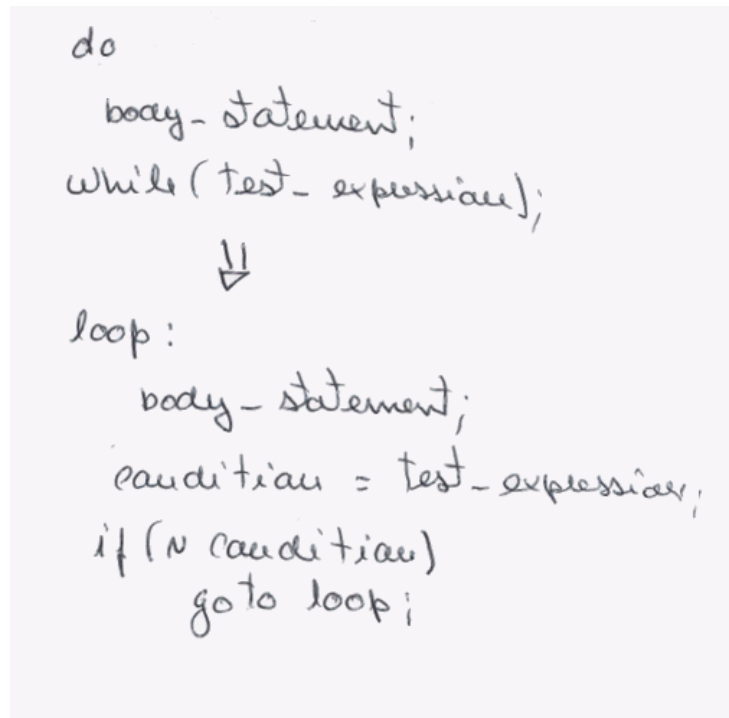
**Figura 2.2:** Algoritmo para if

```

while (test-expression)
    body;
    ⇓
loop:
    condition = test-expression;
    if (¬ condition)
        goto done;
    body-statement;
    goto loop;
done:

```

**Figura 2.3:** Algoritmo para ciclo while



**Figura 2.4:** Algoritmo para ciclo `do while`

```
typedef struct aux {  
    Type val_type;  
    char *s;  
} Instr;  
typedef struct auxvar {  
    Type val_type;  
    Entry *entry;  
    char *s;  
} Var;
```

```
%union{char * val_string;  
    int val_nro;  
    char* val_id;  
    Instr instr;  
    Var var;}
```



```
%token <val_id>id  
%token <val_nro>num  
%token <val_string>string
```

%token BEGINNING  
%token END  
%token VAR  
%token G  
%token L  
%token GEQ  
%token LEQ  
%token EQ  
%token NEQ  
%token NOT  
%token AND  
%token OR  
%token READ  
%token WRITE  
%token IF  
%token ELSE  
%token WHILE  
%token DO

```
%type<instr>Program
%type<instr>Declarations
%type<instr>Body
%type<instr>InstructionsList
%type<instr>Declaration
%type<instr>DeclarationsList
%type<instr>Else
%type<instr>Constant
%type<instr>Term
%type<instr>ExpAdditiv
%type<instr>ExMultipl
%type<instr>Exp
%type<instr>Atribution
%type<instr>Instruction
```

```
%type<var> Variable
```

```
char    *add_label ( CompoundInstruction cpd )
{
    push_label_stack ( status, cpd );
    return push_label ( status, cpd );
}
void remove_label ( CompoundInstruction cpd )
{
    pop_label ( status, cpd );
    reset_label_stack ( status, cpd );
    pop_label_stack ( status, cpd );
}
```

<http://lh3lh3.users.sourceforge.net/udb.shtml>.

# Capítulo 3

## Testes e Resultados

### 3.1 Resultados

#### 3.1.1 Variáveis

---

```
1  VAR x;  
2  
3  
4  BEGINNING  
5  
6  
7    x = y;  
8  
9  
10 END
```

---

---

```
1  VAR x, y;  
2  
3  
4  BEGINNING  
5  
6  
7    x[1] = y;  
8  
9  
10 END
```

---

---

```
1          pushi 0  
2          pushi 0  
3  start  
4  Erro sintático 7: Variável não está declarada em =
```

---

---

```
1  VAR x, x[10], y;  
2  
3  
4  BEGINNING  
5  
6  
7    x[1] = y;  
8  
9  
10 END
```

---

---

```
1          pushi 0  
2 Erro sintático 1: Variável já existe  
3   em ,
```

---

---

```
1  VAR x[10][1], y;  
2  
3  
4  BEGINNING  
5  
6  
7    x = y;  
8  
9  
10 END
```

---

---

```
1          pushn 10  
2          pushi 0  
3  start  
4 Erro sintático 7: Variável não está declarada em =
```

---

---

```
1  VAR x, x[10][1], y;  
2  
3  
4  BEGINNING  
5  
6
```

```
7   x[1][1] = y;  
8  
9  
10  END
```

---

```
1           pushi 0  
2  Erro sintático 1: Variável já existe  
3   em ]
```

---

```
1           pushi 0  
2  Erro sintático 1: Variável já existe  
3   em ,
```

---

```
1  VAR x, x, y;  
2  
3  
4  BEGINNING  
5  
6  
7   x = y;  
8  
9  
10  END
```

---

```
1  VAR x, x, y;  
2  
3  
4  BEGINNING  
5  
6  
7   x = y;  
8  
9  
10  END
```

---



### 3.1.2 Estruturas de Controlo

---

```
1  VAR x, y;  
2  
3  
4  BEGINNING  
5  
6      DO{  
7          DO{  
8              y = x;  
9  
10             }  
11             WHILE ( x > y );  
12         }  
13         WHILE ( x > y );  
14  
15  END
```

---

---

```
1  VAR x, x, y;  
2  
3  
4  BEGINNING  
5  
6  
7      x = y;  
8  
9  
10  END
```

---

---

```
1  VAR x, y;  
2  
3  
4  BEGINNING  
5  
6  
7      IF ( x > y )  
8      {  
9          IF ( x > y )  
10         {  
11             y = x;  
12  
13         }  
14  
15     }
```

```

16         ELSE
17         {
18             y = x;
19
20         }
21
22 END

```

---

```

1         pushi 0
2         pushi 0
3 start
4         pushg 0
5         pushg 1
6         sup
7         jz then11
8         pushg 0
9         pushg 1
10        sup
11        jz then11
12        pushg 0
13        storeg 1
14 then11:      nop
15             jump else1
16 then11:      nop
17             pushg 0
18             storeg 1
19 else1:      nop
20 stop

```

---

```

1 VAR x, y, max;
2
3
4 BEGINNING
5
6     WHILE ( x > y )
7     {
8         WHILE ( x > y )
9         {
10             y = x;
11
12         }
13
14     }
15
16 END

```

---

---

```

1      pushi 0
2      pushi 0
3      pushi 0
4  start
5  wloop2:      nop
6      pushg 0
7      pushg 1
8      sup
9      jz wdone2
10 wloop1:      nop
11      pushg 0
12      pushg 1
13      sup
14      jz wdone1
15      pushg 0
16      storeg 1
17      jump wloop1
18 wdone1:      nop
19      jump wloop2
20 wdone2:      nop
21 stop

```

---

### 3.1.3 Estruturas de Controlo — Tipos

---

```

1  VAR x, y;
2
3
4  BEGINNING
5
6
7
8      DO{
9          y = x;
10
11      }WHILE ( y );
12
13
14
15  END

```

---



---

```

1  VAR x, x, y;
2

```

---

```

3
4 BEGINNING
5
6
7   x = y;
8
9
10  END

```

---

```

1  VAR x, y;
2
3
4  BEGINNING
5
6
7
8      IF ( x ) {
9
10         y = x;
11
12     }
13
14
15
16  END

```

---

```

1      pushi 0
2      pushi 0
3  start
4  Erro sintático 16: A condição não tem um valor booleano em END

```

---

```

1  VAR x, y;
2
3
4  BEGINNING
5
6
7
8      WHILE ( x ) {
9
10         y = x;
11

```

```

12         }
13
14
15
16 END

```

---

```

1         pushi 0
2         pushi 0
3 start
4 Erro sintático 12: A condição não tem um valor booleano em }

```

---

### 3.1.4 Expressões

```

1 VAR x, y;
2
3
4 BEGINNING
5
6
7 IF (x OR (x < 1)) {
8     x=y;
9 }
10
11
12 END

```

---

```

1         pushi 0
2         pushi 0
3 start
4 Erro sintático 7: A expressão não tem elementos do mesmo tipo em )

```

---

```

1 VAR x, y;
2
3
4 BEGINNING
5
6
7 if (x + 1 * (x < 1)) {
8     x=y;
9 }

```

```
10
11
12 END
```

---

```
1      pushi 0
2      pushi 0
3 start
4 Erro sintático 7: Variável não está declarada em (
```

---

```
1 VAR x, y;
2
3
4 BEGINNING
5
6
7 IF ( x AND (x > y) )
8   x=y;
9 }
10
11
12 END
```

---

```
1      pushi 0
2      pushi 0
3 start
4 Erro sintático 7: A expressão não tem elementos do mesmo tipo em )
```

---

```
1 VAR x, y;
2
3
4 BEGINNING
5
6
7 IF (x * 2 / (x > y) ) {
8   x=y;
9 }
10
11
12 END
```

---

---

```
1      pushi 0
2      pushi 0
3  start
4  Erro sintático 7: syntax error em 2
```

---

---

```
1  VAR x, y;
2
3
4  BEGINNING
5
6
7  x = (x > y);
8
9
10 END
```

---

---

```
1      pushi 0
2      pushi 0
3  start
4  Erro sintático 7: Os tipo de elementos da atribuição não são iguais em ;
```

---

## 3.2 Alternativas, Decisões e Problemas de Implementação

# Conclusão

O presente relatório expôs a criação da LPIS e compilador requeridos no problema apresentado.

O desenho de uma linguagem de programação funcional exigiu também um aumento no domínio das funcionalidades Yacc. Dado que os requisitos do projeto foram cumpridos de forma eficaz, o resultado final pode ser considerado positivo, apesar de existir sempre espaço para melhorias, tanto na implementação como no desenho da solução.



# Bibliografia

- [1] T. H. Cormen. *Introduction to Algorithms*. The MIT Press, 2n edition edition, 2009. ISBN 0262533057.
- [2] N. Drakos and R. Moore. Bibtex Entry Types, Field Types and Usage Hints, 1993. URL <http://www.openoffice.org/bibliographic/bibtex-defs.pdf>.
- [3] K. Eleftherios and S. C. Nort. Editing graphs with dotty, 1996. URL <http://www.graphviz.org/pdf/dottyguide.pdf>.
- [4] A. Feder. BibTeX, 2006. URL <http://www.bibtex.org/>.
- [5] P. Lehman, P. Kime, A. Boruvka, and J. Wright. The BibLaTeX Package, 2016. URL <http://ftp.eq.uc.pt/software/TeX/macros/latex/contrib/biblatex/doc/biblatex.pdf>.
- [6] M. E. Lesk and E. Schmidt. Lex: a lexical analyzer generator. 1975. URL <http://dinosaur.compilertools.net/lex/index.html>.
- [7] J. Levine. *Flex & Bison*. 2009. ISBN 978-0-596-15597-1.
- [8] J. R. Levine, T. Mason, D. Brown, and T. Niemann. *Lex And Yacc*. O'Reilly, 1992. ISBN 1-56592-000-7.
- [9] T. Niemann. Lex & Yacc Tutorial. URL <http://epaperpress.com/lexandyacc/>.
- [10] H. Refsnes, S. Refsnes, K. Jim Refsnes, and J. Egil Refsnes. *Learn HTML and CSS with W3Schools*. Wiley Publishing, Inc., Indianapolis, Indiana, 2010. ISBN 0470611952.

# **ANEXOS**

# Apêndice A

## Gramática para linguagem criada

1

$\langle \text{Program} \rangle ::= \langle \text{Declarations} \rangle \langle \text{Body} \rangle$

$\langle \text{Body} \rangle ::= \text{'BEGIN'} \langle \text{InstructionList} \rangle \text{'END'}$

$\langle \text{Declaration} \rangle ::= \langle \text{id} \rangle$

|  $\langle \text{id} \rangle \text{'['} \langle \text{num} \rangle \text{'}'}$

|  $\langle \text{id} \rangle \text{'['} \langle \text{num} \rangle \text{'}' } \text{'['} \langle \text{num} \rangle \text{'}'}$

$\langle \text{Declarations} \rangle ::= \text{'VAR'} \langle \text{DeclarationsList} \rangle \text{';'}$

$\langle \text{DeclarationsList} \rangle ::= \langle \text{Declaration} \rangle$

|  $\langle \text{DeclarationsList} \rangle \text{' ,' } \langle \text{Declaration} \rangle$

$\langle \text{Constant} \rangle ::= \langle \text{num} \rangle$

$\langle \text{Factor} \rangle ::= \langle \text{Constant} \rangle$

|  $\langle \text{Variable} \rangle$

|  $\langle \text{id} \rangle \text{'['} \langle \text{ExpAdditiv} \rangle \text{'}' } \text{'['} \langle \text{ExpAdditiv} \rangle \text{'}'}$

|  $\text{'('} \langle \text{Exp} \rangle \text{'}'}$

|  $\text{'NOT'} \langle \text{Exp} \rangle$

$\langle \text{Variable} \rangle ::= \langle \text{id} \rangle$

|  $\langle \text{id} \rangle \text{'['} \langle \text{ExpAdditiv} \rangle \text{'}'}$

|  $\langle \text{id} \rangle \text{'['} \langle \text{ExpAdditiv} \rangle \text{'}' } \text{'['} \langle \text{ExpAdditiv} \rangle \text{'}'}$

$\langle \text{Term} \rangle ::= \langle \text{Factor} \rangle$

|  $\langle \text{Term} \rangle \text{'*'} \langle \text{Factor} \rangle$

|  $\langle \text{Term} \rangle \text{'/'} \langle \text{Factor} \rangle$

|  $\langle \text{Term} \rangle \text{'\%' } \langle \text{Factor} \rangle$

|  $\langle \text{Term} \rangle \text{'AND'} \langle \text{Factor} \rangle$

$\langle \text{ExpAdditiv} \rangle ::= \langle \text{Term} \rangle$

|  $\langle \text{ExpAdditiv} \rangle \text{'+' } \langle \text{Term} \rangle$

|  $\langle \text{ExpAdditiv} \rangle \text{'-' } \langle \text{Term} \rangle$

|  $\langle \text{ExpAdditiv} \rangle \text{'OR'} \langle \text{Term} \rangle$

---

<sup>1</sup>A gramática está na notação *Backus-Naur Form*

$$\begin{aligned}
\langle \text{Exp} \rangle &::= \langle \text{ExpAdditiv} \rangle \\
&| \langle \text{ExpAdditiv} \rangle \text{'>'} \langle \text{ExpAdditiv} \rangle \\
&| \langle \text{ExpAdditiv} \rangle \text{'<'} \langle \text{ExpAdditiv} \rangle \\
&| \langle \text{ExpAdditiv} \rangle \text{'>='} \langle \text{ExpAdditiv} \rangle \\
&| \langle \text{ExpAdditiv} \rangle \text{'<='} \langle \text{ExpAdditiv} \rangle \\
&| \langle \text{ExpAdditiv} \rangle \text{'=='} \langle \text{ExpAdditiv} \rangle \\
&| \langle \text{ExpAdditiv} \rangle \text{'!='} \langle \text{ExpAdditiv} \rangle \\
\\
\langle \text{Else} \rangle &::= \langle \rangle \\
&| \text{'ELSE' } \langle \text{InstructionsList} \rangle \text{' '} \\
\\
\langle \text{Atribution} \rangle &::= \langle \text{Variable} \rangle \text{'='} \langle \text{ExpAdditiv} \rangle \\
\\
\langle \text{Instruction} \rangle &::= \langle \text{Atribution} \rangle \text{' ; ' } \\
&| \text{'READ' } \langle \text{Variable} \rangle \text{' ; ' } \\
&| \text{'WRITE' } \langle \text{ExpAdditiv} \rangle \text{' ; ' } \\
&| \text{'WRITE' } \langle \text{string} \rangle \text{' ; ' } \\
&| \text{'IF' } \text{' ( ' } \langle \text{Exp} \rangle \text{' ) ' } \text{' { ' } } \langle \text{InstructionsList} \rangle \text{' } \text{' } \langle \text{Else} \rangle \\
&| \text{'WHILE' } \text{' \ ( ' } \langle \text{Exp} \rangle \text{' ) ' } \text{' { ' } } \langle \text{InstructionsList} \rangle \text{' } \text{' } \\
&| \text{'DO' } \text{' { ' } } \langle \text{InstructionsList} \rangle \text{' } \text{' } \text{'WHILE' } \text{' \ ( ' } \langle \text{Exp} \rangle \text{' ) ' } \text{' ; ' } \\
\\
\langle \text{InstructionList} \rangle &::= \langle \text{Instruction} \rangle \\
&| \langle \text{InstructionList} \rangle \langle \text{Instruction} \rangle
\end{aligned}$$

# Apêndice B

## Código do analisador léxico do *Flex*

---

```
1  %{
2
3  %}
4
5  letra [A-Za-z]
6  digito [0-9]
7  lixo .|\n
8  %option yylineno
9  %%
10
11  [-/;,\[\] +\(\)\{\}\%]=]      {return (yytext[0]);}
12  (AND)                        {return AND;}
13  (OR)                         {return OR;}
14  (<)                          {return L;}
15  (>)                          {return G;}
16  (<=)                         {return LEQ;}
17  (>=)                         {return GEQ;}
18  (==)                         {return EQ;}
19  (!=)                         {return NEQ;}
20  (NOT)                         {return NOT;}
21  (BEGINNING)                  {return BEGINNING;}
22  (END)                         {return END;}
23  (READ)                       {return READ;}
24  (WRITE)                      {return WRITE;}
25  (VAR)                        {return VAR;}
26  (WHILE)                      {return WHILE;}
27  (IF)                         {return IF;}
28  (ELSE)                       {return ELSE;}
29  (DO)                         {return DO;}
30  "[^"]+"                      {yylval.val_string = strdup(yytext); return
    ↪ string;}
31  {digito}+                    {yylval.val_nro = atoi(yytext); return num;}
32  {letra}+                     {yylval.val_id= strdup(yytext); return id;}
33  {lixo}                       {;}
34
```

```
35  %%  
36  
37  int yywrap() {  
38      return 1;  
39  }
```

---

**Listing 1:** Código do analisador léxico

# Apêndice C

## Código do analisador sintático e tradutor do YACC

---

```
1  // *INDENT-OFF*
2  %{
3  // *INDENT-ON*
4  #define _GNU_SOURCE
5  #include <stdio.h>
6  #include <string.h>
7  #include <stdlib.h>
8  #include "../src/program_status.h"
9  #include "../src/entry.h"
10 #include "../src/types.h"
11     int yylex();
12     int yylineno;
13     typedef struct aux {
14         Type val_type;
15         char *s;
16     } Instr;
17     typedef struct auxvar {
18         Type val_type;
19         Entry *entry;
20         char *s;
21     } Var;
22     Program_status *status = NULL;
23     char *add_label ( CompoundInstruction cpd )
24     {
25         push_label_stack ( status, cpd );
26         return push_label ( status, cpd );
27     }
28     void remove_label ( CompoundInstruction cpd )
29     {
30         pop_label ( status, cpd );
31         reset_label_stack ( status, cpd );
32         pop_label_stack ( status, cpd );
33     }
```

```

34     int yyerror ( char *s );
35 // *INDENT-OFF*
36 %}
37 %union{char * val_string; int val_nro; char* val_id; Instr instr; Var var;}
38 %token <val_id>id
39 %token <val_nro>num
40 %token <val_string>string
41 %token BEGINNING
42 %token END
43 %token VAR
44 %token G
45 %token L
46 %token GEQ
47 %token LEQ
48 %token EQ
49 %token NEQ
50 %token NOT
51 %token AND
52 %token OR
53 %token READ
54 %token WRITE
55 %token IF
56 %token ELSE
57 %token WHILE
58 %token DO
59 %type<instr>Program
60 %type<instr>Declarations
61 %type<instr>Body
62 %type<instr>InstructionsList
63 %type<instr>Declaration
64 %type<instr>DeclarationsList
65 %type<instr>Else
66 %type<instr>Constant
67 %type<instr>Factor
68 %type<instr>ExpAdditiv
69 %type<instr>Term
70 %type<instr>Exp
71 %type<instr>Atribution
72 %type<instr>Instruction
73 %type<var> Variable
74 %start Program
75 %%
76 Program : Declarations Body      {printf("%s", $2.s);}
77 ;
78 Body : BEGINNING {printf("start\n");} InstructionsList END
    ⇐ {asprintf(&$$.s,
79     "%sstop\n", $3.s);}
80 ;

```



```

81 Declaration : id
82 {
83 // *INDENT-ON*
84     Entry *entry = find_identifier ( status, $1 );
85
86     if ( entry == NULL )
87     {
88         printf ( "\tpushi 0\n" );
89         add_Variable ( status, $1, Integer, Variable, Program );
90         $$ .val_type = Integer;
91     }
92
93     else
94     {
95         yyerror( "Variável já existe\n" );
96         exit ( -1 );
97     }
98
99 // *INDENT-OFF*
100 }
101 | id '[' num ']'
102 {
103 // *INDENT-ON*
104     Entry *entry = find_identifier ( status, $1 );
105
106     if ( entry==NULL )
107     {
108         add_Array ( status, $1, Integer, Array, $3, Program );
109         printf ( "\tpushn %d\n", $3 );
110         $$ .val_type = Integer;
111     }
112
113     else
114     {
115         yyerror( "Variável já existe\n" );
116         exit ( -1 );
117     }
118
119 // *INDENT-OFF*
120 }
121 | id '[' num ']' '[' num ']'
122 {
123 // *INDENT-ON*
124     Entry *entry = find_identifier ( status, $1 );
125
126     if ( entry==NULL )
127     {
128         add_Matrix ( status, $1, Integer, Matrix, $3*$6, $3, Program );

```

```

129     printf ( "\tpushn %d\n", $3*$6 );
130     $$ .val_type = Integer;
131 }
132
133 else
134 {
135     yyerror( "Variável já existe\n" );
136     exit ( -1 );
137 }
138
139 // *INDENT-OFF*
140 }
141 ;
142 Declarations : VAR DeclarationsList ';'      { $$ .s=$2.s; }
143 ;
144 DeclarationsList : Declaration              { $$ .s=$1.s; }
145 | DeclarationsList ',' Declaration        { asprintf(&$$ .s, "%s%s", $1.s,
    ↪     $3.s); }
146 ;
147
148 Variable : id
149 {
150     // *INDENT-ON*
151     Entry *entry = find_identifier ( status, $1 );
152
153     if ( entry && get_class(entry) == Variable)
154     {
155         //int address = get_address ( entry );
156         //asprintf ( &$$ .s, "\t " );
157         $$ .val_type = Integer;
158         $$ .entry = entry;
159     }
160
161     else
162     {
163         yyerror( "Variável não está declarada" );
164         exit ( -1 );
165     }
166
167 // *INDENT-OFF*
168 }
169 | id '[' ExpAdditiv ']'
170 {
171     // *INDENT-ON*
172     Entry *entry = find_identifier ( status, $1 );
173
174     if ( entry && get_class(entry) == Array)
175     {

```

```

176     int address = get_address ( entry );
177     asprintf ( &$$$.s, "\tpushgp\n\tpushg %d\n\tpadd\n%s",address, $$$3.s
↪ );
178     $$$$.val_type=Integer;
179     $$$$.entry=entry;
180 }
181
182 else
183 {
184     yyerror( "Variável não está declarada" );
185     exit ( -1 );
186 }
187
188 // *INDENT-OFF*
189 }
190 | id '[' ExpAdditiv ']' '[' ExpAdditiv ']' {
191 // *INDENT-ON*
192     Entry *entry = find_identifier ( status, $$$1 );
193
194     if ( entry&&get_class(entry)==Matrix)
195     {
196         int address = get_address ( entry );
197         int nRows = get_nRows ( entry );
198         asprintf ( &$$$$.s, "\tpushgp\n\tpushg %d\n\tpadd\n\tpushi
↪ %d\n%s\tml\n%s\tadd\n",
199                 address, nRows, $$$3.s, $$$6.s );
200         $$$$.val_type=Integer;
201         $$$$.entry=entry;
202     }
203
204     else
205     {
206         yyerror( "Variável não está declarada" );
207         exit ( -1 );
208     }
209
210 // *INDENT-OFF*
211 }
212 ;
213 Constant : num {
214 // *INDENT-ON*
215     asprintf ( &$$$$.s, "\tpushi %d\n", $$$1 );
216
217     $$$$.val_type = Integer;
218
219 // *INDENT-OFF*
220 }
221 ;

```

```

222 Factor : Constant
223 {
224 // *INDENT-ON*
225     $$$.s=$1.s;
226
227     $$$.val_type=$1.val_type;
228 }
229 | Variable {
230     // *INDENT-ON*
231     if ( get_class ( $1.entry ) == Matrix || get_class ( $1.entry ) == Array
    ↪ )
232     {
233         asprintf ( &$$$.s, "%s\tloadn\n", $1.s );
234     }
235
236     else {
237         int address = get_address ( $1.entry );
238         asprintf ( &$$$.s, "\tpushg %d\n", address );
239     }
240     //$$$.s=$1.s;
241     $$$.val_type=$1.val_type;
242 // *INDENT-OFF*
243 }
244 | '-' (' Exp' )
245 {
246 // *INDENT-ON*
247     if ( check_type ( $3.val_type, Integer ) )
248     {
249         asprintf ( &$$$.s, "%s\tpushi -1\n\tsub\n", $3.s );
250         $$$.val_type=$3.val_type;
251     }
252
253     else {
254         yyerror( "A condição não tem um valor inteiro" );
255         exit ( -1 );
256     }
257 // *INDENT-OFF*
258 }
259 | '(' Exp ')'
260 {
261 // *INDENT-ON*
262     $$$.s=$2.s;
263
264     $$$.val_type=$2.val_type;
265
266 // *INDENT-OFF*
267 }
268 | NOT '(' Exp ')'

```

```

269 {
270 // *INDENT-ON*
271     if ( check_type ( $3.val_type, Boolean ) )
272     {
273         $$ .val_type=$3.val_type;
274         $$ .s=$3.s;
275     }
276
277     else {
278         yyerror( "A condição não tem um valor booleano" );
279         exit ( -1 );
280     }
281 // *INDENT-OFF*
282 }
283 ;
284 Term : Factor
285 {
286     $$ .s=$1.s;
287     $$ .val_type=$1.val_type;
288 }
289 | Term '*' Factor
290 {
291 // *INDENT-ON*
292     if ( check_type ( $1.val_type, Integer ) &&check_type ( $3.val_type,
↪ Integer ) )
293     {
294         asprintf ( &$$ .s, "%s%s\tml\n", $1.s, $3.s );
295         $$ .val_type=$1.val_type;
296     }
297
298     else {
299         yyerror( "A expressão não tem elementos do mesmo tipo " );
300         exit ( -1 );
301     }
302 // *INDENT-OFF*
303 }
304 | Term '/' Factor
305 {
306 // *INDENT-ON*
307     if ( check_type ( $1.val_type, Integer ) &&check_type ( $3.val_type,
↪ Integer ) )
308     {
309         asprintf ( &$$ .s, "%s%s\tdiv\n", $1.s, $3.s );
310         $$ .val_type=$1.val_type;
311     }
312
313     else {
314         yyerror( "A expressão não tem elementos do mesmo tipo " );

```

```

315         exit ( -1 );
316     }
317     // *INDENT-OFF*
318 }
319 | Term '%' Factor
320 {
321     // *INDENT-ON*
322     if ( check_type ( $1.val_type, Integer ) &&check_type ( $3.val_type,
↪ Integer ) )
323     {
324         asprintf ( &$$.s, "%s%s\tmod\n", $1.s, $3.s );
325         $$val_type=$1.val_type;
326     }
327
328     else {
329         yyerror( "A expressão não tem elementos do mesmo tipo " );
330         exit ( -1 );
331     }
332     // *INDENT-OFF*
333 }
334 | Term AND Factor
335 {
336     // *INDENT-ON*
337     if ( check_type ( $1.val_type, Boolean ) &&check_type ( $3.val_type,
↪ Boolean ) )
338     {
339         asprintf ( &$$.s, "%s%s\tmul\n", $1.s, $3.s );
340         $$val_type=Boolean;
341     }
342
343     else {
344         yyerror( "A expressão não tem elementos do mesmo tipo " );
345         exit ( -1 );
346     }
347     // *INDENT-OFF*
348 }
349 ;
350 ExpAdditiv : Term
351 {
352     // *INDENT-ON*
353     $$s=$1.s;
354
355     $$val_type=$1.val_type;
356
357     // *INDENT-OFF*
358 }
359 | ExpAdditiv '+' Term
360 {

```

```

361 // *INDENT-ON*
362     if ( check_type ( $1.val_type, Integer ) &&check_type ( $3.val_type,
↪ Integer ) )
363     {
364         asprintf ( &$$s, "%s%s\tadd \n", $1.s, $3.s );
365         $$val_type=$1.val_type;
366     }
367
368     else {
369         yyerror( "A expressão não tem elementos do mesmo tipo " );
370         exit ( -1 );
371     }
372 // *INDENT-OFF*
373 }
374 | ExpAdditiv '-' Term
375 {
376 // *INDENT-ON*
377     if ( check_type ( $1.val_type, Integer ) &&check_type ( $3.val_type,
↪ Integer ) )
378     {
379         asprintf ( &$$s, "%s%s\tsub \n", $1.s, $3.s );
380         $$val_type=$1.val_type;
381     }
382
383     else {
384         yyerror( "A expressão não tem elementos do mesmo tipo " );
385         exit ( -1 );
386     }
387 // *INDENT-OFF*
388 }
389 | ExpAdditiv OR Term
390 {
391 // *INDENT-ON*
392     if ( check_type ( $1.val_type, Boolean ) &&check_type ( $3.val_type,
↪ Boolean ) )
393     {
394         asprintf ( &$$s, "%s%s\tadd \n", $1.s, $3.s );
395         $$val_type=Boolean;
396     }
397
398     else {
399         yyerror( "A expressão não tem elementos do mesmo tipo " );
400         exit ( -1 );
401     }
402 // *INDENT-OFF*
403 }
404 ;
405 Exp : ExpAdditiv

```

```

406 {
407 // *INDENT-ON*
408 $$$.s=$1.s;
409
410 $$$.val_type=$1.val_type;
411
412 // *INDENT-OFF*
413 }
414 | ExpAdditiv L    ExpAdditiv
415 {
416 // *INDENT-ON*
417     if ( check_type ( $1.val_type, Integer ) &&check_type ( $3.val_type,
↪ Integer ) )
418     {
419         asprintf ( &$$$$.s, "%s%s\tinf \n", $1.s, $3.s );
420         $$$$.val_type=Boolean;
421     }
422
423     else {
424         yyerror( "A expressão não tem elementos do mesmo tipo " );
425         exit ( -1 );
426     }
427 // *INDENT-OFF*
428 }
429 | ExpAdditiv G    ExpAdditiv
430 {
431 // *INDENT-ON*
432     if ( check_type ( $1.val_type, Integer ) &&check_type ( $3.val_type,
↪ Integer ) )
433     {
434         asprintf ( &$$$$.s, "%s%s\tsup \n", $1.s, $3.s );
435         $$$$.val_type=Boolean;
436     }
437
438     else {
439         yyerror( "A expressão não tem elementos do mesmo tipo " );
440         exit ( -1 );
441     }
442 // *INDENT-OFF*
443 }
444 | ExpAdditiv GEQ ExpAdditiv
445 {
446 // *INDENT-ON*
447     if ( check_type ( $1.val_type, Integer ) &&check_type ( $3.val_type,
↪ Integer ) )
448     {
449         asprintf ( &$$$$.s, "%s%s\tsupeq\n", $1.s, $3.s );
450         $$$$.val_type=Boolean;

```



```

451     }
452
453     else {
454         yyerror( "A expressão não tem elementos do mesmo tipo " );
455         exit ( -1 );
456     }
457 // *INDENT-OFF*
458 }
459 | ExpAdditiv LEQ ExpAdditiv
460 {
461 // *INDENT-ON*
462     if ( check_type ( $1.val_type, Integer ) &&check_type ( $3.val_type,
↪ Integer ) )
463     {
464         asprintf ( &$$$.s, "%s%s\tnfeq\n", $1.s, $3.s );
465         $$$$.val_type=Boolean;
466     }
467
468     else {
469         yyerror( "A expressão não tem elementos do mesmo tipo " );
470         exit ( -1 );
471     }
472 // *INDENT-OFF*
473 }
474 | ExpAdditiv EQ ExpAdditiv
475 {
476 // *INDENT-ON*
477     if ( check_type ( $1.val_type, Integer ) &&check_type ( $3.val_type,
↪ Integer ) )
478     {
479         asprintf ( &$$$.s, "%s%s\tequal\n", $1.s, $3.s );
480         $$$$.val_type=Boolean;
481     }
482
483     else {
484         yyerror( "A expressão não tem elementos do mesmo tipo " );
485         exit ( -1 );
486     }
487 // *INDENT-OFF*
488 }
489 | ExpAdditiv NEQ ExpAdditiv
490 {
491 // *INDENT-ON*
492     if ( check_type ( $1.val_type, Integer ) &&check_type ( $3.val_type,
↪ Integer ) )
493     {
494         asprintf ( &$$$.s, "%s%s\tequal\n\tnot\n", $1.s, $3.s );
495         $$$$.val_type=Boolean;

```

```

496     }
497
498     else {
499         yyerror( "A expressão não tem elementos do mesmo tipo " );
500         exit ( -1 );
501     }
502 // *INDENT-OFF*
503 }
504 ;
505 Attribution : Variable '=' ExpAdditiv {
506 // *INDENT-ON*
507     if ( check_type ( $1.val_type, Integer ) && check_type ( $3.val_type,
↪ Integer ) )
508     {
509         if ( get_class ( $1.entry ) == Matrix || get_class ( $1.entry ) ==
↪ Array )
510         {
511             asprintf ( &$$s, "%s%s\tstore\n", $1.s, $3.s );
512         }
513
514         else {
515             int address = get_address ( $1.entry );
516             asprintf ( &$$s, "%s\tstoreg %d\n", $3.s, address );
517         }
518     } else {
519         yyerror( "Os tipo de elementos da atribuição não são iguais" );
520         exit ( -1 );
521     }
522
523
524 // *INDENT-OFF*
525 }
526 ;
527 InstructionsList : Instruction { $$s=$1.s; }
528 | InstructionsList Instruction { asprintf(&$$s, "%s%s", $1.s,
↪ $2.s); }
529 ;
530 Else : {
531 // *INDENT-ON*
532
533     ///pop_label ( status,if_inst );
534
535     asprintf ( &$$s, "then%s:\tnop\n", get_label ( status,if_inst ) );
536
537
538
539 // *INDENT-OFF*
540 }

```

```

541 |
542 ELSE '{' InstructionsList '}' {
543 // *INDENT-ON*
544     char *tmp1 = add_label ( else_inst );
545
546     //char *tmp2 = get_label ( status,if_inst );
547     char *tmp2 = get_label ( status, if_inst );
548
549     //remove_label ( if_inst );
550
551
552     char *tmp = get_label ( status,else_inst );
553
554     asprintf ( &$$$.s, "\tjump else%s\nthen%s:\tnop\n%selse%s:\tnop\n", tmp1,
555 tmp2, $3.s, tmp );
556
557     remove_label ( else_inst );
558
559 // *INDENT-OFF*
560 }
561 Instruction : Attribution ';' {$$$$.s=$1.s;}
562 | READ Variable ';'
563 {
564     if ( get_class ( $2.entry ) == Matrix || get_class ( $2.entry ) == Array
565 ↪ )
566     {
567         asprintf ( &$$$$.s, "%s\tread\n\tatoi\n\tstoren\n", $2.s);
568     }
569
570     else {
571         int address = get_address ( $2.entry );
572         asprintf ( &$$$$.s, "\tread\n\tatoi\n\tstoreg %d\n", address);
573     }
574 // *INDENT-OFF*
575 }
576 | WRITE ExpAdditiv ';'
577 {
578 // *INDENT-ON*
579     if ( check_type ( $2.val_type, Integer ) )
580     {
581         asprintf ( &$$$$.s, "%s\twritei\n", $2.s );
582     }
583
584     else {
585         yyerror( "Não é possível escrever valores booleanos" );
586         exit ( -1 );
587     }
588 // *INDENT-OFF*

```

```

588 }
589 | WRITE string ';'
590 {
591 // *INDENT-ON*
592     asprintf ( &$.s, "\tpushs %s\n\twrites\n", $2 );
593
594 // *INDENT-OFF*
595 }
596 | IF { add_label( if_inst); } '(' Exp ')' '{' InstructionsList '}'
597 Else
598 {
599 // *INDENT-ON*
600     if ( check_type ( $.val_type, Boolean ) )
601     {
602         asprintf ( &$.s, "%s\tjz then%s\n%s%s", $.s, get_label
603         ( status, if_inst), $.s, $.s );
604         //printf ( "\tjz then%s\n", add_label ( if_inst ) );
605         //asprintf ( &$<instr>$.s, "%s", $3.s );
606         //printf ( "\tjz then%s\n", add_label ( if_inst ) );
607         remove_label ( if_inst );
608     }
609
610     else {
611         yyerror( "A condição não tem um valor booleano");
612         exit ( -1 );
613     }
614
615     //asprintf ( &$.s, "%s%s", $.s, $.s );
616
617 // *INDENT-OFF*
618 }
619 | WHILE '(' Exp ')' '{' InstructionsList '}'
620 {
621 // *INDENT-ON*
622     if ( check_type ( $.val_type, Boolean ) )
623     {
624         char *tmp = add_label ( while_inst );
625         asprintf ( &$.s, "wloop%s:\tnop\n%s\tjz wdone%s\n%s\tjump
↪ wloop%s\nwdone%s:\tnop\n", tmp, $.s, tmp, $.s, tmp, tmp );
626         remove_label ( while_inst );
627
628     } else {
629         yyerror( "A condição não tem um valor booleano" );
630         exit ( -1 );
631     }
632 // *INDENT-OFF*
633 }
634 | DO '{' InstructionsList '}' WHILE '(' Exp ')' ';'

```

```

635
636 {
637 // *INDENT-ON*
638     if ( check_type ( $7.val_type, Boolean ) )
639     {
640         char *tmp = add_label ( do_while_inst );
641         asprintf ( &$$.s, "doloop%s:\tnop\n%s%\tjz dodone%\tjump
↪ doloop%s\n\tdodone%s:\tnop\n\n", tmp, $3.s, $7.s, tmp,tmp, tmp );
642         remove_label ( do_while_inst );
643
644     } else {
645         yyerror( "A condição não tem um valor booleano" );
646         exit ( -1 );
647     }
648 // *INDENT-OFF*
649 }
650 ;
651 %%
652 // *INDENT-ON*
653 #include "lex.yy.c"
654 int yyerror ( char *mensagem )
655 {
656
657     printf ( "Erro sintático %d: %s em %s\n", yylineno, mensagem, yytext);
658     return 0;
659 }
660
661 int main()
662 {
663     status = ( Program_status * ) malloc ( sizeof ( struct stat ) );
664     status = init ( status );
665
666     if ( status==NULL )
667         return -1;
668
669     push_label_stack ( status, if_inst );
670     push_label_stack ( status, else_inst );
671     push_label_stack ( status, while_inst );
672     push_label_stack ( status, do_while_inst );
673     yyparse();
674     pop_label_stack ( status, if_inst );
675     pop_label_stack ( status, else_inst );
676     pop_label_stack ( status, while_inst );
677     pop_label_stack ( status, do_while_inst );
678     return 0;
679 }

```

---

**Listing 2:** Código do analisador sintático

# Apêndice D

## Código gerado a partir do tradutor do *YACC*, para os exemplos pedidos

### D.1 Maior de três números

#### D.1.1 LPIS

---

```
1  VAR x, y, max;
2
3
4  BEGINNING
5
6  WRITE "escreva x:";
7  READ x;
8  WRITE "escreva y:";
9  READ y;
10
11 IF ( x > y )
12 {
13     max = x;
14
15 }
16 ELSE
17 {
18     max = y;
19
20 }
21
22 WRITE "O maior numero e:";
23 WRITE max;
24
25 END
```

---

## D.1.2 VM

---

```
1      pushi 0
2      pushi 0
3      pushi 0
4  start
5      pushs "escreva x:"
6      writes
7      read
8      atoi
9      storeg 0
10     pushs "escreva y:"
11     writes
12     read
13     atoi
14     storeg 1
15     pushg 0
16     pushg 1
17     sup
18     jz then1
19     pushg 0
20     storeg 2
21     jump else1
22  then1:      nop
23             pushg 1
24             storeg 2
25  else1:      nop
26             pushs "O maior numero e:"
27             writes
28             pushg 2
29             writei
30  stop
```

---

## D.2 Somatório de N números

### D.2.1 LPIS

---

```
1  VAR n, current, sum, counter;
2
3  BEGINNING
4
5      WRITE "Escreva o total de numeros :";
6      READ n;
7
8      current = 0;
9      sum = 0;
```

```

10         counter = 0;
11
12         WHILE (counter < n)
13         {
14             WRITE "Escreva um numero:";
15             READ current;
16             sum = sum + current;
17
18             counter = counter +1;
19
20         }
21         WRITE "O valor total da soma e:";
22         WRITE sum;
23
24     END

```

---

## D.2.2 VM

---

```

1         pushi 0
2         pushi 0
3         pushi 0
4         pushi 0
5     start
6         pushs "Escreva o total de numeros :"
7         writes
8         read
9         atoi
10        storeg 0
11        pushi 0
12        storeg 1
13        pushi 0
14        storeg 2
15        pushi 0
16        storeg 3
17    wloop1:    nop
18        pushg 3
19        pushg 0
20        inf
21        jz wdone1
22        pushs "Escreva um numero:"
23        writes
24        read
25        atoi
26        storeg 1
27        pushg 2
28        pushg 1
29        add

```



```

30         storeg 2
31         pushg 3
32         pushi 1
33         add
34         storeg 3
35         jump wloop1
36 wdone1:      nop
37         pushs "O valor total da soma e:"
38         writes
39         pushg 2
40         writei
41 stop

```

---

## D.3 Sequência de pares de N números dados

### D.3.1 LPIS

---

```

1  VAR current, counter;
2
3  BEGINNING
4
5  current = 1;
6  counter = 0;
7
8
9  WHILE (current != 0)
10 {
11
12     WRITE "Escreva um numero:";
13     READ current;
14
15     IF(current % 2 == 0)
16     {
17         WRITE current;
18         counter = counter +1;
19     }
20
21
22
23
24 }
25
26 WRITE "Total de numeros pares lidos:";
27 WRITE counter;
28
29 END

```

---

### D.3.2 VM

---

```
1      pushi 0
2      pushi 0
3  start
4      pushi 1
5      storeg 0
6      pushi 0
7      storeg 1
8  wloop1:      nop
9      pushg 0
10     pushi 0
11     equal
12     not
13     jz wdone1
14     pushs "Escreva um numero:"
15     writes
16     read
17     atoi
18     storeg 0
19     pushg 0
20     pushi 2
21     mod
22     pushi 0
23     equal
24     jz then1
25     pushg 0
26     writei
27     pushg 1
28     pushi 1
29     add
30     storeg 1
31  then1:      nop
32     jump wloop1
33  wdone1:      nop
34     pushs "Total de numeros pares lidos:"
35     writes
36     pushg 1
37     writei
38  stop
```

---

## D.4 Ordenação de *array* de tamanho N–*Insertion Sort*

### D.4.1 LPIS

---

```

1  VAR a[20], pos, n, x;
2
3  BEGINNING
4      pos = 0;
5      READ n;
6
7      WHILE (pos < n)
8      {
9          READ x;
10         a[pos] = x;
11
12         pos = pos +1;
13
14     }
15
16     pos = 1;
17
18
19     WHILE ( pos < n )
20     {
21         IF (a[pos - 1] <= a[pos])
22         {
23             pos = pos +1;
24
25         }
26         ELSE
27         {
28             x = a[pos];
29             a[pos] = a[pos - 1];
30             a[pos - 1] = x;
31
32             pos = pos - 1;
33
34             IF( pos == 0)
35             {
36                 pos = 1;
37             }
38
39         }
40
41     }
42     pos = 0;
43     WHILE ( pos < n )
44     {
45         WRITE a[pos];
46
47     }
48 END

```

---

## D.4.2 VM

---

```
1      pushn 20
2      pushi 0
3      pushi 0
4      pushi 0
5  start
6      pushi 0
7      storeg 20
8      read
9      atoi
10     storeg 21
11  wloop1:      nop
12     pushg 20
13     pushg 21
14     inf
15     jz wdone1
16     read
17     atoi
18     storeg 22
19     pushgp
20     pushg 0
21     padd
22     pushg 20
23     pushg 22
24     storen
25     pushg 20
26     pushi 1
27     add
28     storeg 20
29     jump wloop1
30  wdone1:      nop
31     pushi 1
32     storeg 20
33  wloop2:      nop
34     pushg 20
35     pushg 21
36     inf
37     jz wdone2
38     pushgp
39     pushg 0
40     padd
41     pushg 20
42     pushi 1
43     sub
44     loadn
45     pushgp
46     pushg 0
```

```

47      padd
48      pushg 20
49      loadn
50      infeq
51      jz then11
52      pushg 20
53      pushi 1
54      add
55      storeg 20
56      jump else1
57  then11:      nop
58      pushgp
59      pushg 0
60      padd
61      pushg 20
62      loadn
63      storeg 22
64      pushgp
65      pushg 0
66      padd
67      pushg 20
68      pushgp
69      pushg 0
70      padd
71      pushg 20
72      pushi 1
73      sub
74      loadn
75      storen
76      pushgp
77      pushg 0
78      padd
79      pushg 20
80      pushi 1
81      sub
82      pushg 22
83      storen
84      pushg 20
85      pushi 1
86      sub
87      storeg 20
88      pushg 20
89      pushi 0
90      equal
91      jz then11
92      pushi 1
93      storeg 20
94  then11:      nop

```

```

95  else1:          nop
96          jump wloop2
97  wdone2:         nop
98          pushi 0
99          storeg 20
100 wloop3:         nop
101          pushg 20
102          pushg 21
103          inf
104          jz wdone3
105          pushgp
106          pushg 0
107          padd
108          pushg 20
109          loadn
110          writei
111          jump wloop3
112 wdone3:         nop
113 stop

```

---

## D.5 Média e máximo de uma matriz [N] [M]

### D.5.1 LPIS

---

```

1  VAR matriz [10] [5], media,
2  restomedia, total, max, i, j, n , m, tmp, cont;
3
4  BEGINNING
5      max = 0;
6      media = 0;
7      total = 0;
8      tmp = 0;
9      cont = 0;
10     WRITE "escreva o numero de linhas :";
11     READ n;
12     WRITE "escreva o numero de colunas :";
13     READ m;
14
15     i = 0;
16
17     WHILE (i < n)
18     {
19
20         j = 0;
21         WHILE (j < m)
22         {

```

```

23         READ tmp;
24
25         matriz[i][j] = tmp;
26         total = total + tmp;
27         cont = cont + 1;
28
29         IF(tmp > max)
30         {
31             max = tmp;
32
33         }
34
35         j = j + 1;
36     }
37
38     i = i + 1;
39 }
40
41 media = total/cont;
42 restomedia = total % cont;
43
44 WRITE "A media tem um valor de :";
45 WRITE media;
46 WRITE "o resto tem um valor de :";
47 WRITE restomedia;
48 WRITE "O maximo tem um valor de :";
49 WRITE max;
50 END

```

---

## D.5.2 VM

---

```

1         pushn 50
2         pushi 0
3         pushi 0
4         pushi 0
5         pushi 0
6         pushi 0
7         pushi 0
8         pushi 0
9         pushi 0
10        pushi 0
11        pushi 0
12 start
13        pushi 0
14        storeg 53
15        pushi 0
16        storeg 50

```

```

17      pushi 0
18      storeg 52
19      pushi 0
20      storeg 58
21      pushi 0
22      storeg 59
23      pushs "escreva o numero de linhas :"
24      writes
25      read
26      atoi
27      storeg 56
28      pushs "escreva o numero de colunas :"
29      writes
30      read
31      atoi
32      storeg 57
33      pushi 0
34      storeg 54
35  wloop2:      nop
36      pushg 54
37      pushg 56
38      inf
39      jz wdone2
40      pushi 0
41      storeg 55
42  wloop1:      nop
43      pushg 55
44      pushg 57
45      inf
46      jz wdone1
47      read
48      atoi
49      storeg 58
50      pushgp
51      pushg 0
52      padd
53      pushi 10
54      pushg 54
55      mul
56      pushg 55
57      add
58      pushg 58
59      storen
60      pushg 52
61      pushg 58
62      add
63      storeg 52
64      pushg 59

```



```

65         pushi 1
66         add
67         storeg 59
68         pushg 58
69         pushg 53
70         sup
71         jz then1
72         pushg 58
73         storeg 53
74 then1:      nop
75         pushg 55
76         pushi 1
77         add
78         storeg 55
79         jump wloop1
80 wdone1:     nop
81         pushg 54
82         pushi 1
83         add
84         storeg 54
85         jump wloop2
86 wdone2:     nop
87         pushg 52
88         pushg 59
89         div
90         storeg 50
91         pushg 52
92         pushg 59
93         mod
94         storeg 51
95         pushs "A media tem um valor de :"
96         writes
97         pushg 50
98         writei
99         pushs "o resto tem um valor de :"
100        writes
101        pushg 51
102        writei
103        pushs "O maximo tem um valor de :"
104        writes
105        pushg 53
106        writei
107 stop

```

---