Processamento de Linguagens (3º ano de Curso)

Trabalho Prático N 2

Relatório de Desenvolvimento

Filipe Costa Oliveira
a57816

30 de Maio de 2016

Conteúdo

Capítulo 1

Introdução

O presente trabalho prático foca-se no desenvolvimento de um compilador, que tem como fonte uma linguagem de alto nível (também esta desenvolvida especificamente para este trabalho prático) , gerando código para uma máquina de stack virtual.

Um compilador comum divide o processo de tradução em várias fases. Para o propósito específico desta unidade curricular iremos focar-nos nas seguintes:

- 1ª Fase de tradução – Análise Léxica, que agrupa sequências de caracteres em tokens. Recorreremos nesta fase à definição das expressões regulares que permitem definir os tokens.
- 2ª Fase de tradução – Reconhecimento(Parsing) da estrutura gramatical do programa, através do agrupamento dos tokens em produções. Recorreremos à definição de uma gramática independente de contexto por forma a definir as estruturas de programa válidas a reconhecer pelo parser. Denote que juntamente com o parsing é realizada a análise semântica, assim como a geração de código associando regras às produções anteriormente descritas.

Começaremos portanto por definir uma linguagem de programação imperativa simples, que chamaremos Algebra. A Algebra permitirá:

- declarar e manusear variáveis atómicas do tipo inteiro, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas.
- 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. Aos arrays de duas dimensões, por se tratar de uma linguagem algébrica, chamaremos matrizes, dada a fácil associação a este tipo de variável à sua definição análoga da álgebra linear.
- efetuar instruções algorítmicas básicas como a atribuição de expressões a variáveis.
- ler do standard input e escrever no standard output.
- efetuar instruções para controlo do fluxo de execução – condicional e cíclica – que possam ser aninhadas.
- definir e invocar subprogramas sem parâmetros mas que possam retornar um resultado atómico.

Na nossa linguagem de programação por questões de estruturação e percepção, teremos como premissa que as varíaveis deverão ser declaradas no início do programa, não podendo haver re-declarações, nem utilizações sem declaração prévia. Não será permitida a declaração e associação de um valor inteiro na mesma instrução. Achamos essa solução pouco elegante. Assim, todas as variáveis terão o valor zero após a declaração.

Será desenvolvido portanto o compilador para a Algebra, com base na GIC criada acima e recurso ao Gerador Yacc/ Flex. O compilador de Algebra irá gerar pseudo-código, Assembly da Máquina Virtual VM cuja documentação completa está disponibilizada em anexo.

Por forma a facilitar e validar o trabalho, à medida que as funcionalidades forem descritas serão apensados exemplos ilustrativos.
Por fim, serão apresentados um conjunto de testes mais complexos(programas-fonte diversos e respectivo código produzido), que tentam testar de uma forma mais alargadas as funcionalidades da Algebra, sendo estes:

- lidos 3 números, escrever o maior deles.
- ler N (valor dado) números e calcular e imprimir o seu somatório.
- contar e imprimir os números pares de uma sequência de N números dados.
- ler e armazenar os elementos de um vetor de comprimento N, imprimido os valores por ordem crescente após fazer a ordenação do array por trocas diretas.
- ler e armazenar os elementos de uma matriz NxM, calculando e imprimindo de seguida a média e máximo dessa matriz.
- invocar e usar num programa uma função.

Capítulo 2

Concepção da Linguagem Algebra

2.1 Concepção/desenho da Resolução

Comecemos por descrever as funcionalidades da linguagem Algebra. Tal como descrito anteriormente a Algebra permitirá:

- declarar e manusear variáveis atómicas do tipo inteiro, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas.
- 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. Aos arrays de duas dimensões, por se tratar de uma linguagem algébrica, chamaremos matrizes, dada a fácil associação a este tipo de variável à sua definição análoga da álgebra linear.
- efetuar instruções algorítmicas básicas como a atribuição de expressões a variáveis.
- ler do standard input e escrever no standard output.
- efetuar instruções para controlo do fluxo de execução – condicional e cíclica – que possam ser aninhadas.
- definir e invocar subprogramas sem parâmetros mas que possam retornar um resultado atómico.

2.1.1 Uma introdução às variáveis

Temos então que as variáveis poderão ser de 3 tipos: inteiros simples, arrays de inteiros, e matrizes de inteiros. Dessa premissa sabemos à partida que o código gerado para a nossa máquina virtual terá que suportar o tipo de variável inteiro. Sabemos ainda que aos tipos de dados mais complexos (arrays e matrizes) apenas é permita a realização de operações de indexação.

Na nossa linguagem de programação por questões de estruturação e percepção, teremos como premissa que as variáveis deverão ser declaradas no início do programa, não podendo haver re-declarações, nem utilizações sem declaração prévia. Não será permitida a declaração e associação de um valor inteiro na mesma instrução(à lá C). Achamos essa solução pouco elegante. Assim, todas as variáveis terão o valor zero após a declaração.

Podemos então aceitar como exemplo as declarações do tipo:

```
1 int a;
2 int auxiliar_1;
3 int array_1d[10];
4 int exemplo_2d[40,2];
```

Dado que toda a porção de código de alto nível julgamos essencial a possibilidade de existência de comentários. Atente no exemplo anterior agora com comentários que facilitam a percepção:

```
// variaveis do tipo inteiro
int a;
int auxiliar_1;
// arrays de inteiros
int array_1d[10];
// matrizes
int exemplo_2d[40,2];
```

Tal como poderá confirmar pela última declaração do exemplo anterior a declaração do tamanho das matrizes é feita da seguinte forma: **nome_variavel[nºlinhas,nºcolunas]**.

A forma de armazenamento e acesso às variáveis será posteriormente discutida nas secções seguintes deste relatório. Neste momento temos especial interesse na especificação da estrutura correcta de programas da nossa linguagem.

2.1.1.1 Expressões Regulares e acções resultantes

Podemos desde já enumerar as expressões regulares necessárias à produção dos tokens que permitam à GIC o agrupamento dos tokens em produções:

```
%{

%}

letter      [a-zA-Z]
digit       [0-9]
ignore      [\ \t\r\n]

%option yylineno

%%

[\%\,\{\}\+\-\(\)\=\>\<\!\;\/\*\[\]\|\&\_]      { return(yytext[0]); }
int                                             { return (TYPE_INT); }

{letter}({letter}|{digit}|\_)*                  { yylval.var = strdup(yytext); return(id); }
{digit}+                                        { yylval.qt = atoi(yytext); return(num); }
\/\/[^\n]*                                      { printf("%s\n",yytext);  }
{ignore}                                        { ; }

%%

int yywrap(){
  return(1);
}
```

Como é perceptível pela expressão regular correspondente, vulgo {letter}({letter}|{digit}|_)*, as variáveis do tipo inteiro terão sempre de ser iniciadas por uma letra (maiúscula ou minúscula), sendo que como segundo caracter poderão ter um número, letra, ou _.

2.1.1.2 Produções da GIC

Com os tokens produzidos pelo parser, podemos iniciar a definição da gramática independente de contexto, resultando nas seguintes produções:

```
%{

%}

%union {int qt; char* var;}

%token <var>id
%token <qt>num

%token TYPE_INT
```

```
11
12 %nonassoc PL_THEN
13 %nonassoc PL_ELSE
14
15 %start AlgebricScript
16
17 %%
18
19 AlgebricScript : Declarations
20                ;
21
22 Declarations  :
23               Declarations Declaration ';'
24               | /*empty*/
25               ;
26
27 Declaration  :
28              TYPE_INT id
29              | TYPE_INT id '[' num  ',' num  ']'
30              | TYPE_INT id '[' num  ']'
31              ;
32 %%
33
34 #include "lex.yy.c"
35
36 int yyerror(char* s) {
37     if (strlen(yytext)>1){
38         printf("\t\terr \"Error (input file line %d): %s at %s\"\n", yylineno, s, yytext);
39         fprintf(stderr,"Error\t (line %d): %s at %s\n", yylineno, s, yytext);
40     }
41     else {
42         printf("\t\terr \"Error (input file line %d): %s\"\n", yylineno, s);
43         fprintf(stderr,"Error\t (line %d): %s\n", yylineno, s);
44     }
45     return 1;
46 }
47
48 int main () {
49     yyparse();
50     return 0;
51 }
```

2.1.2 Uma introdução às instruções

Da necessidade de realizar operações aritméticas, relacionais e lógicas sobre as variáveis do tipo inteiro atómicas, assim como da necessidade de realizar instruções algorítmicas básicas como a atribuição de expressões a variáveis, surgem as **instruções** na nossa linguagem ***Algebra***.

Consideramos que qualquer que seja a operação a ser realizada, o seu resultado terá que ser sempre atribuído a alguma variável.

Podemos desde já enumerar os tipos de operações permitidas na nossa linguagem, associando também o operador utilizador para representar as mesmas:

- Aritmética
  - Adição : '+'
  - Subtração : '-'
  - Multiplicação inteira : '*'
  - Divisão Inteira : '/'
  - Resto da Divisão Inteira : '
- Relaccional

  - Igualdade : '="="'
  - Diferença : '!"='
  - Superioridade : '>'
  - Superioridade ou Igualdade : '>"='
  - Inferioridade : '<'
  - Inferioridade ou Igualdade : '<"='
- Lógica
  - Negação Lógica : '!'
  - OR Lógico : '|"|'
  - AND Lógico : '&&'

Podemos então aceitar como exemplo de input válido o seguinte código:

```
1 // declaracoes iniciais
2 int a;
3 int b;
4 int c;
5
6 // operacoes de atribuicao
7 a = 7;
8 b = 3;
9
10 // operacoes aritmeticas
11 c = 1 + b*a / 2;
```

Como poderá constatar pelas linhas 7 e 8, e tal como é requerido já será possível realizar operações de atribuição.

2.1.2.1 Expressões Regulares e acções resultantes

Relativamente às expressões regulares necessárias para proceder correctamente ao parsing não é necessário alterar os ficheiro Flex presente na seção 2.1.1.1, uma vez que a expressão regular [\%\,\{\}\+\-\(\)\=\>\<\!\;\/*\[\]\|\&_] já engloba todos os símbolos necessárias até à fase actual.

2.1.2.2 Produções da GIC

Tomando por base o ficheiro Yacc presente na seção 2.1.1.2, podemos proceder à adição de produções por forma a reconhecer as estruturas de programa válidas até ao momento.

```
1 %{
2
3 %}
4
5 %union {int qt; char* var;}
6
7 %token <var>id
8 %token <qt>num
9
10 %token TYPE_INT
11
12 %start AlgebricScript
13
14 %%
15
16 AlgebricScript : Declarations Instructions
17                ;
18
19 Declarations  :
20               Declarations Declaration ';'
```

```
21                | /*empty*/
22                ;
23
24 Declaration   :
25                TYPE_INT id
26                | TYPE_INT id '[' num ',' num ']'
27                | TYPE_INT id '[' num ']'
28                ;
29
30 Instructions : Instructions Instruction
31                | /*empty*/
32                ;
33
34 Instruction : Assignment ';'
35              ;
36
37 Assignment : id '=' Assignement_Value
38             | Vectors '=' Assignement_Value
39             ;
40
41 Assignement_Value : Arithmetic_Expression
42                    ;
43
44 Vectors : id '[' Arithmetic_Expression Second_Dimension Dimension_End
45         ;
46
47 Second_Dimension : ',' Arithmetic_Expression
48                   | /*empty*/
49                   ;
50
51 Dimension_End : ']'
52                ;
53
54 Arithmetic_Expression : Term
55                        | Arithmetic_Expression '+' Term
56                        | Arithmetic_Expression '-' Term
57                        ;
58
59 Term     : Factor
60          | Term '*' Factor
61          | Term '/' Factor
62          | Term '%' Factor
63          ;
64
65 Factor   : num
66          | id
67          | Vectors
68          | '(' Arithmetic_Expression ')'
69          ;
70
71 Logical_Expressions : Logical_Expressions Logical_Expression
72                      |
73                      ;
74
75 Logical_Expression : '!' Relational_Expression
76                    | Relational_Expression
77                    | Logical_Expression '|''|' Relational_Expression
78                    | Logical_Expression '&''&' Relational_Expression
79                    ;
80
81 Relational_Expression : Arithmetic_Expression
82                        | Arithmetic_Expression '=''=' Arithmetic_Expression
83                        | Arithmetic_Expression '!''=' Arithmetic_Expression
84                        | Arithmetic_Expression '>' Arithmetic_Expression
85                        | Arithmetic_Expression '>''=' Arithmetic_Expression
86                        | Arithmetic_Expression '<' Arithmetic_Expression
87                        | Arithmetic_Expression '<''=' Arithmetic_Expression
88                        | '(' Logical_Expressions ')'
```

```
                        ;
%%

#include "lex.yy.c"

int yyerror(char* s) {
    if (strlen(yytext)>1){
        printf("\t\terr \"Error (input file line %d): %s at %s\"\n", yylineno, s, yytext);
        fprintf(stderr,"Error\t (line %d): %s at %s\n", yylineno, s, yytext);
    }
    else {
        printf("\t\terr \"Error (input file line %d): %s\"\n", yylineno, s);
        fprintf(stderr,"Error\t (line %d): %s\n", yylineno, s);
    }
    return 1;
}

int main () {
    yyparse();
    return 0;
}
```

As produções relativas às expressões lógicas e relacionais terão especial importância na adição da capacidade de inclusão de instruções para controlo do fluxo de execução – condicional e cíclica – que possam ser aninhadas, na nossa linguagem ***Algebra***, que passaremos de seguida e especificar.

2.1.3 Uma introdução às instruções condicionais e cíclicas

2.1.3.1 Instruções condicionais

Por forma à ***Algebra*** ter utilidade real, é necessária a inclusão de instruções que permitam mudar o fluxo de execução. Necessitamos portanto de incluir a possibilidade de declarar instruções condicionais na nossa linguagem.

Para criarmos uma estrutura condicional, deveremos recorrer a expressões do tipo:

if (Expressão Lógica)
then [{Instruções}|Instrução]
else [{Instruções}|Instrução|/*empty*/]

Pela análise do esquema anterior sabemos que o bloco de código **else** [{Instruções}|Instrução|/*empty*/] é opcional, sendo que, em caso de os fluxos de execução representarem apenas uma instrução na nossa linguagem ***Algebra*** não existe a necessidade de inclusão de parêntesis entre os diferentes fluxos.

Tal como requerido, deverá ser também possível o aninhamento de instruções condicionais.

Podemos então aceitar como exemplo de input válido o seguinte código:

```
// declaracoes inciais
int a;
int b;
int c;
int maior;

// operacoes de atribuicao
a = 15;
b = 7 * 4;
c = 120 % 1;

// instrucoes condicionais
if ( a >= b && a >= c ) then {
  maior = a;
}
else {
  if ( b > a && b >= c  ) then {
    maior = b;
  }
```

```
20    else {
21      // esta condicao era desnecessaria
22      // mas desta forma provamos o correcto aninhamento de condicionais
23      if (c > a && c > b ) then {
24        maior = c;
25      }
26      // este condicional nao tem o fluxo else
27    }
28  }
```

2.1.3.2 Instruções cíclicas

Uma instrução cíclica irá permitir ao programador executar um determinado bloco de código um determinado número de vezes, de acordo com uma condição lógica.

Para criarmos uma estrutura cíclica, deveremos recorrer a expressões do tipo:

do [{Instruções}|Instrução]
while (Expressão Lógica)

Pela análise do esquema anterior sabemos que em caso de o fluxo de execução representar apenas uma instrução na nossa linguagem ***Algebra*** não existe a necessidade de inclusão de parêntesis entre as palavras reservadas **do** e **while**.

Tal como requerido, deverá ser também possível o aninhamento de instruções condicionais.

Podemos então aceitar como exemplo de input válido o seguinte código:

```
1  // declaracoes inciais
2  int a;
3  int b;
4  int c;
5  int maior;
6
7  // operacoes de atribuicao
8  a = 1;
9  a = b; // estamos a atribuir directamente a b o valor de a
10 c = 20 % 1;
11
12 // instrucoes ciclicas
13 do {
14   do {
15     a = a + 1;
16   }
17   while ( a < c )
18   b = b + 1;
19 }
20 while ( b < c )
```

2.1.3.3 Expressões Regulares e acções resultantes

Tomando por base o ficheiro Flex presente na seção 2.1.1.1, podemos proceder à adição de expressões regulares por forma a produzir os tokens necessários para o correcto reconhecimento pela GIC.

```
1  %{
2
3  %}
4
5  letter      [a-zA-Z]
6  digit       [0-9]
7  ignore      [\ \t\r\n]
8
9  %option yylineno
10
11 %%
```

```
12
13 [\%\,\{\}\+\-\(\)\=\>\<\!\;\/\*\[\]\|\&\-]      { return(yytext[0]); }
14 do                                              { return (PL_DO); }
15 while                                           { return (PL_WHILE); }
16 if                                              { return (PL_IF); }
17 then                                            { return (PL_THEN); }
18 else                                            { return (PL_ELSE); }
19 int                                             { return (TYPE_INT); }
20
21 {letter}({letter}|{digit}|\_)*                  { yylval.var = strdup(yytext); return(id); }
22 {digit}+                                        { yylval.qt = atoi(yytext); return(num); }
23 \/\/[^\n]*                                      { printf("%s\n",yytext);  }
24 {ignore}                                        { ; }
25
26 %%
27
28 int yywrap(){
29   return(1);
30 }
```

2.1.3.4 Produções da GIC

Tomando por base o ficheiro Yacc presente na seção 2.1.1.2, podemos proceder à adição de produções por forma a reconhecer as estruturas de programa válidas até ao momento.

```
1 %{
2
3 %}
4
5 %union {int qt; char* var;}
6
7 %token <var>id
8 %token <qt>num
9 %token <var>string
10
11 %token TYPE_INT
12
13 %token PL_IF PL_THEN PL_ELSE
14 %token PL_DO PL_WHILE
15
16 %start AlgebricScript
17
18 %%
19
20 AlgebricScript : Declarations Instructions
21                ;
22
23 Declarations  :
24               Declarations Declaration ';'
25               | /*empty*/
26               ;
27
28 Declaration   :
29              TYPE_INT id
30              | TYPE_INT id '[' num  ',' num  ']'
31              | TYPE_INT id '[' num  ']'
32              ;
33
34 Instructions : Instructions Instruction
35              | /*empty*/
36              ;
37
38 Instruction : Assignment ';'
39             | Conditional
40             | Cycle
41             ;
```

```
42
43 Assignment : id '=' Assignement_Value
44            | Vectors '=' Assignement_Value
45            ;
46
47 Assignement_Value : Arithmetic_Expression
48                   ;
49 Vectors : id
50         '['
51         Arithmetic_Expression
52         Second_Dimension Dimension_End
53         ;
54
55 Second_Dimension : ','  Arithmetic_Expression
56                  | /*empty*/
57                  ;
58
59 Dimension_End : ']'
60               ;
61
62 Arithmetic_Expression : Term
63                       | Arithmetic_Expression '+' Term
64                       | Arithmetic_Expression '-' Term
65                       ;
66
67 Term     : Factor
68         | Term '*' Factor
69         | Term '/' Factor
70         | Term '%' Factor
71         ;
72
73 Factor   : num
74         | id
75         | Vectors
76         | '(' Arithmetic_Expression ')'
77         ;
78
79 Logical_Expressions : Logical_Expressions Logical_Expression
80                     |
81                     ;
82
83 Logical_Expression : '!' Relational_Expression
84                    | Relational_Expression
85                    | Logical_Expression '|''|' Relational_Expression
86                    | Logical_Expression '&''&' Relational_Expression
87                    ;
88
89 Relational_Expression : Arithmetic_Expression
90                       | Arithmetic_Expression '=''=' Arithmetic_Expression
91                       | Arithmetic_Expression '!''=' Arithmetic_Expression
92                       | Arithmetic_Expression '>' Arithmetic_Expression
93                       | Arithmetic_Expression '>''=' Arithmetic_Expression
94                       | Arithmetic_Expression '<' Arithmetic_Expression
95                       | Arithmetic_Expression '<''=' Arithmetic_Expression
96                       | '(' Logical_Expressions ')'
97                       ;
98
99 Conditional : If_Starter PL_THEN '{' Instructions '}' Else_Clause
100            | If_Starter PL_THEN  Instruction  Else_Clause
101            ;
102
103 If_Starter : PL_IF '(' Logical_Expressions ')'
104            ;
105
106 Else_Clause : PL_ELSE '{' Instructions '}'
107             | PL_ELSE Instruction
108             | /*empty*/
109             ;
```

```
110
111 Cycle : PL_DO  '{' Instructions '}' PL_WHILE '(' Logical_Expressions ')'
112        | PL_DO  Instruction PL_WHILE '(' Logical_Expressions ')'
113        ;
114
115 %%
116
117 #include "lex.yy.c"
118
119 int yyerror(char* s) {
120     if (strlen(yytext)>1){
121         printf("\t\terr \"Error (input file line %d): %s at %s\"\n", yylineno, s, yytext);
122         fprintf(stderr,"Error\t (line %d): %s at %s\n", yylineno, s, yytext);
123     }
124     else {
125         printf("\t\terr \"Error (input file line %d): %s\"\n", yylineno, s);
126         fprintf(stderr,"Error\t (line %d): %s\n", yylineno, s);
127     }
128     return 1;
129 }
130
131 int main () {
132     yyparse();
133     return 0;
134 }
```

2.1.4 Uma introdução às instruções de leitura do standard input e escrita no standard output

2.1.4.1 Instruções de leitura do standard input

Por forma à ***Algebra*** poder efectuar operações de leitura do standard input, é necessária a inclusão de instruções que permitam a leitura de inteiros, e atribuição do valor inteiro a uma variável.

Consideramos que só fará sentido ler dados do standard input se os mesmos foram atribuídos. "Ler por Ler"do Standard Input não representa nenhuma mais valia para a linguagem.

Podemos então aceitar como exemplo de input válido o seguinte código:

```
1  // declaracoes iniciais
2  int a;
3  int b;
4  int c;
5  int maior;
6
7  // leitura do standard input
8  // e atribuicao do valor lido a variaveis
9  a = read();
10 b = read();
11 c = read();
12
13 if ( a >= b && a >= c ) then {
14   maior = a;
15 }
16 else {
17   if ( b > a && b >= c  ) then {
18     maior = b;
19   }
20   else {
21     // esta condicao era desnecessaria
22     // mas desta forma provamos o correcto aninhamento de condicionais
23     if (c > a && c > b ) then {
24       maior = c;
25     }
26   }
27 }
```

2.1.4.2 Instruções de escrita no standard output

Uma instrução de escrita no standard output permitirá ao programador imprimir o valor de variáveis do tipo inteiro atómicas, variáveis do tipo array e matriz, assim como de valores inteiros directamente, e ainda de variáveis do tipo string, vulgo uma sequência de caracteres iniciada por "e terminada por ".

Denote que na nossa linguagem não é permitido realizar operações sobre strings (como concatenação ou comparação), apenas será permitir escrever as mesmas no standard output.

Ora, retomando o exemplo da seção 2.1.4.1, podemos agora torná-lo mais completo com adição de instruções de escrita no standard input.

```
// declaracoes iniciais
int a;
int b;
int c;
int maior;

// leitura do standard input
// e atribuicao do valor lido a variaveis
print "a: "; // escrita no standard output de uma string
a = read();
print "b: "; // escrita no standard output de uma string
b = read();
print "c: "; // escrita no standard output de uma string
c = read();

print "maior: "; // escrita no standard output de uma string
if ( a >= b && a >= c ) then {
  maior = a;
}
else {
  if ( b > a && b >= c  ) then {
    maior = b;
  }
  else {
    // esta condicao era desnecessaria
    // mas desta forma provamos o correcto aninhamento de condicionais
    if (c > a && c > b ) then {
      maior = c;
    }
  }
}

// escrita no standard output de uma variavel do tipo inteiro atomica
print maior;
```

2.1.4.3 Expressões Regulares e acções resultantes

Tomando por base o ficheiro Flex presente na seção 2.1.3.3, podemos proceder à adição de expressões regulares por forma a produzir os tokens necessários para o correcto reconhecimento pela GIC.

```
%{

%}

letter      [a-zA-Z]
digit       [0-9]
ignore      [\ \t\r\n]

%option yylineno

%%

[\%\,\{\}\+\-\(\)\=\>\<\!\;\/\*\[\]\|\&\.]      { return(yytext[0]); }
do                                              { return (PL_DO); }
```

```
15 while                                          { return (PL_WHILE); }
16 if                                             { return (PL_IF); }
17 then                                           { return (PL_THEN); }
18 else                                           { return (PL_ELSE); }
19 int                                            { return (TYPE_INT); }
20 print                                          { return (PL_PRINT);}
21 read                                           { return (PL_READ);}
22
23 {letter}({letter}|{digit}|\_)*                 { yylval.var = strdup(yytext); return(id); }
24 {digit}+                                       { yylval.qt = atoi(yytext); return(num); }
25 \"[^"]+\"----------------------------------------{ yylval.var = strdup(yytext); return(string); }
26 \/\/[^\n]*----------------------------------------{ printf("%s\n",yytext); }
27 {ignore}----------------------------------------{ ; }
28
29 %%
30
31 int yywrap(){
32 --return(1);
33 }
```

2.1.4.4 Produções da GIC

Tomando por base o ficheiro Yacc presente na seção 2.1.3.4, podemos proceder à adição de produções por forma a reconhecer as estruturas de programa válidas até ao momento.

```
1  %{
2
3  %}
4
5  %union {int qt; char* var;}
6
7  %token <var>id
8  %token <qt>num
9  %token <var>string
10
11 %token TYPE_INT
12
13 %token PL_IF PL_THEN PL_ELSE
14 %token PL_DO PL_WHILE
15 %token PL_PRINT
16 %token PL_READ
17
18 %start AlgebricScript
19
20 %%
21
22 AlgebricScript : Declarations Instructions
23                ;
24
25 Declarations  :
26               Declarations Declaration ';'
27               | /*empty*/
28               ;
29
30 Function_Declarations :
31                       Function_Declarations Function_Declaration
32                       | /*empty*/
33                       ;
34
35 Declaration  :
36              TYPE_INT id
37              | TYPE_INT id '[' num ',' num ']'
38              | TYPE_INT id '[' num ']'
39              }
40              ;
41
```

```
42 Return_Statement : PL_RETURN Arithmetic_Expression ';'
43                  ;
44
45 Instructions : Instructions Instruction
46              | /*empty*/
47              ;
48
49 Instruction : Assignment ';'
50             | WriteStdout ';'
51             | Conditional
52             | Cycle
53             ;
54
55 Assignment : id '=' Assignement_Value
56            | Vectors '=' Assignement_Value
57            ;
58
59 Assignement_Value : Arithmetic_Expression
60                   | Read_Stdin
61                   ;
62 Vectors : id
63         '['
64         Arithmetic_Expression
65         Second_Dimension Dimension_End
66         ;
67
68 Second_Dimension : ',' Arithmetic_Expression
69                  | /*empty*/
70                  ;
71
72 Dimension_End : ']'
73               ;
74
75 Read_Stdin : PL_READ '(' ')'
76            ;
77
78 Arithmetic_Expression : Term
79                       | Arithmetic_Expression '+' Term
80                       | Arithmetic_Expression '-' Term
81                       ;
82
83 Term     : Factor
84         | Term '*' Factor
85         | Term '/' Factor
86         | Term '%' Factor
87         ;
88
89 Factor   : num
90         | id
91         | Vectors
92         | '(' Arithmetic_Expression ')'
93         ;
94
95 Logical_Expressions : Logical_Expressions Logical_Expression
96                     |
97                     ;
98
99 Logical_Expression : '!' Relational_Expression
100                   | Relational_Expression
101                   | Logical_Expression '|''|' Relational_Expression
102                   | Logical_Expression '&''&' Relational_Expression
103                   ;
104
105 Relational_Expression : Arithmetic_Expression
106                       | Arithmetic_Expression '=''=' Arithmetic_Expression
107                       | Arithmetic_Expression '!''=' Arithmetic_Expression
108                       | Arithmetic_Expression '>' Arithmetic_Expression
109                       | Arithmetic_Expression '>''=' Arithmetic_Expression
```

```
110                         | Arithmetic_Expression '<' Arithmetic_Expression
111                         | Arithmetic_Expression '<''=' Arithmetic_Expression
112                         | '(' Logical_Expressions ')'
113                         ;
114
115 Conditional : If_Starter PL_THEN '{' Instructions '}' Else_Clause
116             | If_Starter PL_THEN  Instruction Else_Clause
117             ;
118
119 If_Starter :
120            PL_IF
121            '(' Logical_Expressions ')'
122            ;
123
124 Else_Clause : PL_ELSE '{' Instructions '}'
125             | PL_ELSE Instruction
126             | /*empty*/
127             ;
128
129 Cycle : PL_DO '{' Instructions '}' PL_WHILE '(' Logical_Expressions ')'
130       | PL_DO  Instruction PL_WHILE '(' Logical_Expressions ')'
131       ;
132
133 WriteStdout : PL_PRINT id
134             | PL_PRINT Vectors
135             | PL_PRINT num
136             | PL_PRINT string
137             ;
138 %%
139
140 #include "lex.yy.c"
141
142 int yyerror(char* s) {
143     if (strlen(yytext)>1){
144         printf("\t\terr \"Error (input file line %d): %s at %s\"\n", yylineno, s, yytext);
145         fprintf(stderr,"Error\t (line %d): %s at %s\n", yylineno, s, yytext);
146     }
147     else {
148         printf("\t\terr \"Error (input file line %d): %s\"\n", yylineno, s);
149         fprintf(stderr,"Error\t (line %d): %s\n", yylineno, s);
150     }
151     return 1;
152 }
153
154 int main () {
155     yyparse();
156     return 0;
157 }
```

Possuímos neste momento todas as ferramentas necessárias para a escrita de programas na linguagem de alto nível ***Algebra*** que recorram aos seguintes requisitos:

- declarar e manusear variáveis atómicas do tipo inteiro, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas.
- declarar e manusear variáveis estruturadas do tipo array e matrizes de inteiros, em relação aos quais é apenas permitida a operação de indexação.
- efetuar instruções algorítmicas básicas como a atribuição de expressões a variáveis.
- ler do standard input e escrever no standard output.
- efetuar instruções para controlo do fluxo de execução – condicional e cíclica – que possam ser aninhadas.

Resta-nos passar à adição de funcionalidades que permitam definir e invocar subprogramas sem parâmetros mas que possam retornar um resultado atómico.

2.1.5 Uma introdução aos subprogramas

Por forma a definir e invocar subprogramas sem parâmetros mas que possam retornar um resultado atómico, é necessária a inclusão de instruções que permitam a declaração de blocos de código que apenas serão executados quando invocada a função onde estes estão inscritos.

Ora, mantendo a premissa de estruturação e percepção já existente, decidimos que as declarações de funções terão de ocorrer depois das declarações de variáveis, e antes do início do programa principal.

Não será permitida redeclaração de funções, nem utilização de funções no programa principal sem declaração prévia.

Entendemos que especificamente para a na nossa linguagem de alto nível, todos os subprogramas retornam sempre uma variável com valor inteiro. Tendo em conta o descrito anteriormente, para criarmos um subprograma, deveremos recorrer a instruções do tipo:

declare nome_função ( ) {
Instruções
return valor_inteiro ; }

Para invocarmos um subprograma, deveremos a instruções do tipo:

variável = **call** nome_função ( ) ;

2.1.5.1 Expressões Regulares e acções resultantes

Tomando por base o ficheiro Flex presente na seção 2.1.4.3, podemos proceder à adição de expressões regulares por forma a produzir os tokens necessários para o correcto reconhecimento pela GIC.

```
%{

%}

letter      [a-zA-Z]
digit       [0-9]
ignore      [\ \t\r\n]

%option yylineno

%%

[\%\,\{\}\+\-\(\)\=\>\<\!\;\/\*\[\]\|\&\-]      { return(yytext[0]); }
do                                              { return (PL_DO); }
while                                           { return (PL_WHILE); }
if                                              { return (PL_IF); }
then                                            { return (PL_THEN); }
else                                            { return (PL_ELSE); }
int                                             { return (TYPE_INT); }
declare                                         { return (TYPE_FUNCTION); }
print                                           { return (PL_PRINT);}
read                                            { return (PL_READ);}
call                                            { return (PL_CALL);}
return                                          { return (PL_RETURN);}

{letter}({letter}|{digit}|\_)*                  { yylval.var = strdup(yytext); return(id); }
{digit}+                                        { yylval.qt = atoi(yytext); return(num); }
\"[^"]+\"                                       { yylval.var = strdup(yytext); return(string); }
\/\/[^\n]*                                      { printf("%s\n",yytext); }
{ignore}                                        { ; }

%%

int yywrap(){
  return(1);
}
```

2.1.5.2 Produções da GIC

Tomando por base o ficheiro Yacc presente na seção 2.1.4.4, podemos proceder à adição de produções por forma a reconhecer as estruturas de programa válidas até ao momento.

```
%{

%}

%union {int qt; char* var;}

%token <var>id
%token <qt>num
%token <var>string

%token TYPE_INT
%token TYPE_FUNCTION

%token PL_IF PL_THEN PL_ELSE
%token PL_DO PL_WHILE
%token PL_PRINT
%token PL_READ
%token PL_CALL
%token PL_RETURN

%start AlgebricScript

%%

AlgebricScript : Declarations Function_Declarations Instructions
               ;

Declarations  :
              Declarations Declaration ';'
              | /*empty*/
              ;

Function_Declarations :
                      Function_Declarations Function_Declaration
                      | /*empty*/
                      ;

Declaration  :
             TYPE_INT id
             | TYPE_INT id '[' num  ',' num  ']'
             | TYPE_INT id '[' num  ']'
             ;


Function_Declaration :
                     TYPE_FUNCTION id '('')' '{'
                     Instructions
                     Return_Statement '}'
                     ;

Function_Invocation :
                    PL_CALL id '(' ')'
                    ;

Return_Statement : PL_RETURN Arithmetic_Expression';'
                 ;

Instructions : Instructions Instruction
             | /*empty*/
             ;

Instruction : Assignment ';'
            | WriteStdout ';'
```

```
64              | Conditional
65              | Cycle
66              ;
67
68 Assignment : id '=' Assignement_Value
69            | Vectors '=' Assignement_Value
70            ;
71
72 Assignement_Value : Arithmetic_Expression
73                   | Read_Stdin
74                   | Function_Invocation
75                   ;
76 Vectors : id '[' Arithmetic_Expression Second_Dimension Dimension_End
77         ;
78
79 Second_Dimension : ',' Arithmetic_Expression
80                  | /*empty*/
81                  ;
82
83 Dimension_End : ']'
84               ;
85
86 Read_Stdin : PL_READ '(' ')'
87            ;
88
89 Arithmetic_Expression : Term
90                       | Arithmetic_Expression '+' Term
91                       | Arithmetic_Expression '-' Term
92                       ;
93
94 Term     : Factor
95          | Term '*' Factor
96          | Term '/' Factor
97          | Term '%' Factor
98          ;
99
100 Factor  : num
101         | id
102         | Vectors
103         | '(' Arithmetic_Expression ')'
104         ;
105
106 Logical_Expressions : Logical_Expressions Logical_Expression
107                     |
108                     ;
109
110 Logical_Expression : '!' Relational_Expression
111                    | Relational_Expression
112                    | Logical_Expression '|''|' Relational_Expression
113                    | Logical_Expression '&''&' Relational_Expression
114                    ;
115
116 Relational_Expression : Arithmetic_Expression
117                       | Arithmetic_Expression '=''=' Arithmetic_Expression
118                       | Arithmetic_Expression '!''=' Arithmetic_Expression
119                       | Arithmetic_Expression '>' Arithmetic_Expression
120                       | Arithmetic_Expression '>''=' Arithmetic_Expression
121                       | Arithmetic_Expression '<' Arithmetic_Expression
122                       | Arithmetic_Expression '<''=' Arithmetic_Expression
123                       | '(' Logical_Expressions ')'
124                       ;
125
126 Conditional : If_Starter
127             PL_THEN '{' Instructions '}'
128             Else_Clause
129             | If_Starter
130             PL_THEN  Instruction
131             Else_Clause
```

```
132             ;
133
134 If_Starter :
135             PL_IF
136             '(' Logical_Expressions ')'
137             ;
138
139 Else_Clause : PL_ELSE '{' Instructions '}'
140             | PL_ELSE Instruction
141             | /*empty*/
142             ;
143
144 Cycle : PL_DO '{' Instructions '}' PL_WHILE '(' Logical_Expressions ')'
145       | PL_DO Instruction PL_WHILE '(' Logical_Expressions ')'
146       ;
147
148 WriteStdout : PL_PRINT id
149             | PL_PRINT Vectors
150             | PL_PRINT num
151             | PL_PRINT string
152             ;
153 %%
154
155 int yyerror(char* s) {
156     if (strlen(yytext)>1){
157         printf("\t\terr \"Error (input file line %d): %s at %s\"\n", yylineno, s, yytext);
158         fprintf(stderr,"Error\t(line %d): %s at %s\n", yylineno, s, yytext);
159     }
160     else {
161         printf("\t\terr \"Error (input file line %d): %s\"\n", yylineno, s);
162         fprintf(stderr,"Error\t(line %d): %s\n", yylineno, s);
163     }
164     return 1;
165 }
166
167 int main () {
168     yyparse();
169     return 0;
170 }
```

Concluímos neste momento o estudo sobre a linguagem ***Algebra***, e consequentemente a gramática independente de contexto e analisador léxico.

Capítulo 3

Introdução à Máquina Virtual

Concluída a gramática independente de contexto e o analisador léxico, o próximo passo será dar a conhecer a máquina para a qual pretendemos gera código máquina. Trata-se duma maquina de pilhas, composta duma pilha de execução, duma pilha de chamadas, duma zona de código, de duas heaps e de quatro registos.

A pilha de execução contém valores, que podem ser inteiros, reais ou endereços.

As duas heaps contêm, respectivamente, cadeias de caracteres (strings) e blocos estruturados.

Cada um destes tipos de dados é referenciado por endereços. Cada bloco estruturado contem um certo número de valores (do mesmo tipo dos valores que se podem encontrar na pilha). Um endereço pode apontar para quatro tipos de informação: para código, para a pilha, para um bloco estruturado ou para uma string. Três registos permitam o acesso a diferentes partes da pilha:

- O registo sp (stack pointer) aponta para o topo corrente da pilha. Ele aponta para a primeira célula livre da pilha.
- O registo fp (frame pointer) aponta para o endereço de base das variáveis locais.
- O registo gp contem o endereço de base das variáveis globais.

A máquina possui um registo pc que aponta para a instrução corrente (da zona de código) por executar.

A pilha de chamada permite guardar as chamadas: contém pares de apontadores (i, f). O endereço i guarda o registo de instrução pc e f o registo fp.

3.1 As instruções

As instruções são designadas por um nome e podem aceitar um ou dois parâmetros. Estes podem ser:

- constantes inteiras.
- constantes reais.
- cadeias de caracteres delimitadas por aspas. Estas cadeias de caracteres seguem as mesmas regras de formatação que as cadeias da linguagem C (em particular no que diz respeito aos caracteres especiais como \ ", \ n ou \\ ).
- uma etiqueta simbólica designando uma zona no código.

Para o caso específico da linguagem de alto nível que nos propomos a desenvolver temos especial interesse em instruções que lidem com:

- constantes inteiras.
- cadeias de caracteres delimitadas por aspas. (apenas necessárias para as operações de leitura e escrita)
- uma etiqueta simbólica designando uma zona no código.

3.1.1 Operações de base e assunções

Por forma a gerarmos correctamente código máquina devemos ter em consideração os seguintes pontos:

- As operações aritméticas envolvem os valores do topo e do sub-topo da pilha.

  Neste caso quando a operação envolvida é executada, os dois argumentos são retiradas da pilha (refira-se à secção das convenções para perceber o que é retirar valores da pilha) e o resultado é então empilhado.

- O resultado duma operação de comparação é um inteiro que vale 0 ou 1.
- O inteiro 0 representa o valor booleano falso enquanto o valor 1 representa o valor verdade.

De todas as operações disponíveis, apresentamos de seguida aquelas sobre as quais a GIC irá incluir na geração de código máquina. Atente na separação por tipo de operação:

3.1.1.1 Operações sobre inteiros

Instrução	Argumentos	Descrição
ADD		tira da pilha n e m que devem ser inteiros e empilha o resultado m + n
SUB		tira da pilha n e m que devem ser inteiros e empilha o resultado m - n
MUL		tira da pilha n e m que devem ser inteiros e empilha o resultado m × n
DIV		tira da pilha n e m que devem ser inteiros e empilha o resultado m/n
MOD		tira da pilha n e m que devem ser inteiros e empilha o resultado m mod n
NOT		tira da pilha n que deve ser um inteiro e empilha o resultado n = 0
INF		tira da pilha n e m que devem ser inteiros e empilha o resultado m < n
INFEQ		tira da pilha n e m que devem ser inteiros e empilha o resultado m <=n
SUP		tira da pilha n e m que devem ser inteiros e empilha o resultado m > n
SUPEQ		tira da pilha n e m que devem ser inteiros e empilha o resultado m >= n

3.1.1.2 Operações sobre endereços

Instrução	Argumentos	Descrição
PADD		tira da pilha n que deve ser um inteiro e a que deve ser um endereço e empilha o endereço a + n

3.1.1.3 Igualdade

Instrução	Argumentos	Descrição
EQUAL		tira da pilha n seguido de m que devem ser do mesmo tipo e empilha o resultado de n = m

3.1.1.4 Conversões

Instrução	Argumentos	Descrição
ATOI		retira da pilha o endereço duma string e empilha a sua conversão em inteiro. Tal falha quando a string não representa um inteiro.

3.1.1.5 Manipular dados

Instrução	Argumentos	Descrição
PUSHI	n inteiro	empilha n
PUSHN	n inteiro	empilha n vezes o valor inteiro 0
PUSHS	n string	arquiva n na zona das strings e empilha o endereço
PUSHG	n inteiro	empilha o valor localizado em gp[n]
PUSHGP		empilha o valor do registo gp
LOAD	n inteiro	retira da pilha um endereço a e empilha o valor na pilha ou no heap (dependendo do tipo de a) em a[n]
LOADN		retira da pilha um inteiro n, um endereço a e empilha o valor na pilha ou no heap (dependendo do tipo de a) em a[n]
STOREG	n inteiro	retira um valor da pilha e arquiva-a na pilha em gp[n]
STORE	n inteiro	retira da pilha um valor v e um endereço a, arquiva v em a[n]na pilha ou na heap (dependendo do tipo de a)
STOREN		retira da pilha um valor v, um inteiro n e um endereço a, arquiva v no endereço a[n] na pilha ou na heap

3.1.1.6 Input-Output

Instrução	Argumentos	Descrição
WRITEI		retira um inteiro da pilha e imprime o valor na saída standard
WRITES		retira um endereço de uma string da pilha e imprime a string correspondente na saída standard
READ		lê uma string do teclado (concluída por um "\n") e arquiva esta string (sem o "\n") na heap e coloca (empilha) o endereço na pilha..

3.1.1.7 Operações de controlo

Instrução	Argumentos	Descrição
JUMP	label etiqueta	atribui ao registo pc o endereço no código que corresponde a label (pode ser um inteiro ou um valor simbólico).
JZ	label etiqueta	retira da pilha um valor. Se este for nulo então é atribuido ao registo pc o endereço correspondente à label, incrementa simplesmente pc de 1, caso contrário.
PUSHA	label etiqueta	empilha o endereço de programa correspondente a etiqueta label
CALL		retira da pilha um endereço de programa a, salvaguarda pc e fp na pilha das chamadas, afecta a fp o valor corrente de sp e a pc o valor de a.
RETURN		afecta a sp o valor corrente de fp, restaura da pilha de chamadas os valores de fp e de pc, incrementa pc de 1 por forma a encontrar a instrução a seguir a chamada.

3.1.1.8 Inicialização e fim

Instrução	Argumentos	Descrição
START		Afecta o valor de sp a fp
NOP		não faz nada.
ERR	x string	levanta um erro com a mensagem x.
STOP		pára a execução do programa

3.1.1.9 Operações necessárias e não presentes em instruções da VM

Existem operações lógicas e relacionais que não estão disponíveis na VM, nomeadamente:

- Negação Lógica
- OR Lógico
- AND Lógico
- NOT EQUAL Lógico

No entanto recorrendo a instruções presentes nas tabelas 3.1.1.1 e 3.1.1.5, referentes a operações sobre inteiros e a manipulação de dados, conseguimos obter o comportamento lógico dessas mesmas operações.
Relativamente às instruções de controlo de fluxo seria útil a presença da instrução **jnz** que deveria retirar da pilha um valor e se esse fosse não nulo então seria atribuído ao registo pc o endereço correspondente à label. No entanto esta lacuna é também contornável como veremos nas secções seguintes do relatório.

Capítulo 4

Geração de Código Máquina – de Produções a Assembly

Especificadas as instruções disponíveis na máquina virtual, assim como a sua forma de funcionamento, resta-nos incluir nas produções da gramática independente de contexto a geração de código máquina correspondente.

4.1 Métodos e variáveis auxiliares à geração de código máquina

Por forma a implementar correctamente as funcionalidades propostas, é necessário o conhecimento de alguns dados gerais do programa a ser analisado.

Relativamente às variáveis necessitamos de possuir informação relativamente ao seu tipo (se é inteiro atómico, array, matriz ou função), tamanho total ocupado, dimensões (quando aplicável), e posição relativamente ao global pointer.

Ora tal informação é guardada no array de estruturas var_table, que possui capacidade para armazenar dados relativos a 1000 variáveis. O array **ia[x]**[1] permite de forma rápida saber qual o tamanho total ocupado pela variável presente no índice x da var_table.

É mantido também estado sobre o número de condicionais e ciclos abertos e declarados até ao momento.

De seguida apresentam-se todas a variáveis auxiliares assim como a assinatura das funções às quais se recorre para implementar todas as funcionalidades.

```
1  %{
2  #include <stdio.h>
3
4  typedef enum {PL_INTEGER, PL_ARRAY, PL_MATRIX, PL_FUNCTION} var_type;
5
6  typedef struct {
7    char* varname;
8    var_type type;
9    int value;
10   int** values;
11   int size;
12   int rows;
13   int cols;
14 } datatype;
15
16 // array containg the information about the declared vars and functions
17 datatype var_table[1000];
18 // array associating the number of var to the total space used by it
19 int ia[1001];
20 // current global var index
21 int var_index = 0;
22
23 // array containg the closing cycles order
```

[1] Tal solução foi pensada tendo por base forma de representação de matrizes esparsas CSR, daí advindo o nome da variável.

```
24 int closing_cycles_order[100];
25 // array containg the closing conditionals order
26 int closing_conditionals_order[100];
27
28 // refers to the number of opened cycles
29 int opened_cycles = 0;
30 // refers to the number of opened conditionals
31 int opened_conditionals = 0;
32
33 // refers to the number of declared cycles
34 int number_cycles = 0;
35 // refers to the number of declared conditionals
36 int number_conditionals = 0;
37
38 // refers to the cycle position to close in the closing cycles array
39 int cycle_position_to_close = 0;
40 // refers to the conditional position to close in the closing conditionals array
41 int conditional_position_to_close = 0;
42
43 ////////////////////////////////
44 // start of function signatures
45
46 // var/function insertion
47 void insert_int(char* varname);
48 void insert_array(char* varname, int size);
49 void insert_matrix(char* varname, int rows, int cols);
50 void insert_function ( char* function_name );
51
52 // cycle functions
53 int open_cycle();
54 int close_cycle();
55
56 // conditional functions
57 int open_conditional();
58 int close_conditional();
59 int current_conditional();
60
61 // var lookup functions
62 int lookup_int(char* varname);
63 int lookup_array(char* varname, int pos);
64 int lookup_matrix(char* varname, int row, int col);
65
66 // global variables functions
67 int global_pos(char* varname);
68
69 // vector/matrix related functions
70 int is_vector(char* varname);
71 int get_matrix_ncols(char* varname);
72
73 // error checking / handling functions
74 int exists_var(char* varname, var_type type);
75 void assert_no_redeclared_var( char* varname ,var_type type);
76 void assert_declared_var( char* varname, var_type type);
77 void compile_error( char* message);
78 int yyerror();
79
80 // general
81 int yylex();
82
83 // end of function signatures
84 ////////////////////////////////
85
86 %}
```

À medida que formos recorrendo às funções auxiliares iremos apresentar o respectivo código C.

4.2 Geração de código máquina nas produções

4.2.1 Início e término do programa

Comecemos pela inclusão das instruções **START** e **STOP**. Estas iniciam e param a máquina virtual. Assim sendo, as mesmas serão incluídas na produção:

```
AlgebricScript : Declarations Function_Declarations
               Instructions
               ;
```

Sendo o seguinte código associado à produção:

```
AlgebricScript : Declarations Function_Declarations
               { printf("start\n");}
               Instructions
               { printf("stop\n");}
               ;
```

4.2.2 Declarações de variáveis

Sempre que são reconhecidas as produções de declarações de variáveis é necessário alocar o espaço correspondente às mesmas na stack. Ora, assim sendo, o conjunto de instruções de código máquina que permitirão a correcta declaração de variáveis será incluído nas produções:

```
Declaration  :
             TYPE_INT id
             | TYPE_INT id '[' num ',' num ']'
             | TYPE_INT id '[' num ']'
             ;
```

Sendo o seguinte código associado à produção:

```
Declaration  :
             TYPE_INT id
             {
               assert_no_redeclared_var($2,PL_INTEGER);
               insert_int($2);
             }
             | TYPE_INT id '[' num ',' num ']'
             {
               assert_no_redeclared_var($2,PL_MATRIX);
               insert_matrix($2,$4,$6);
             }
             | TYPE_INT id '[' num ']'
             {
               assert_no_redeclared_var($2,PL_ARRAY);
               insert_array($2, $4);
             }
             ;
```

Ora, o código C adicionado às produções recorre a 5 métodos auxiliares ainda não definidos. De seguida apresentam-se os mesmos:

4.2.2.1 Método auxiliar: void assert_no_redeclared_var(char* varname ,var_type type);

```
void assert_no_redeclared_var( char* varname ,var_type type){
    if ( exists_var(varname, type) ){
        compile_error("re-declaring VAR");
    }
}
```

4.2.2.2 Método auxiliar: void compile_error(char* message);

```
void compile_error( char* message){
    yyerror(message);
    exit(0);
}
```

4.2.2.3 Método auxiliar: void insert_int(char* varname);

```
void insert_int ( char* varname ) {
    var_table[var_index].varname = strdup(varname);
    var_table[var_index].value = 0;
    var_table[var_index].type = PL_INTEGER;
    var_table[var_index].size =1;
    int old_size = ia[var_index];
    var_index++;
    ia[var_index] = old_size + 1;
    printf("\t\tpushi 0\t//%s\n",varname);
}
```

4.2.2.4 Método auxiliar: void insert_array(char* varname, int size);

```
void insert_array ( char* varname, int size ) {
    var_table[var_index].varname = strdup(varname);
    var_table[var_index].value = 0;
    var_table[var_index].type = PL_ARRAY;
    var_table[var_index].size =size;
    var_table[var_index].cols = size;
    int old_size = ia[var_index];
    var_index++;
    ia[var_index] = old_size + size;
    printf("\t\tpushn %d\t//%s[%d]\n",size,varname,size);
}
```

4.2.2.5 Método auxiliar: void insert_matrix(char* varname, int rows, int cols);

```
void insert_matrix ( char* varname, int rows, int cols ) {
    var_table[var_index].varname = strdup(varname);
    var_table[var_index].value = 0;
    var_table[var_index].type = PL_MATRIX;
    var_table[var_index].rows = rows;
    var_table[var_index].cols = cols;
    int size = rows * cols;
    var_table[var_index].size =size;
    int old_size = ia[var_index];
    var_index++;
    ia[var_index] = old_size + size;
    printf("\t\tpushn %d\t//%s[%d][%d] (size %d)\n",size,varname,rows,cols,size);
}
```

4.2.3 Declarações de subprogramas

Sempre que são reconhecidas as produções de declarações de subprogramas é necessário alocar o espaço correspondente ao valor inteiro atómico a retornar na stack na stack.

Relativamente à invocação de funções é necessário também incluir os código máquina e **CALL** e **RETURN**, sendo necessária correcta marcação das zonas do código máquina produzido através de labels.

É garantida a não execução do código da função através da inclusão de um salto para o fim da função antes do conjunto de instruções pertencentes à mesma.

Ora, assim sendo, o conjunto de instruções de código máquina que permitirão a correcta declaração e invocação de subprogramas será incluído nas produções:

```
Function_Declaration :
                      TYPE_FUNCTION id '(')' '{'
                      Instructions Return_Statement
                      '}'
                      ;

Function_Invocation :
                     PL_CALL id '(' ')'
                     ;
```

Sendo o seguinte código associado à produção:

```
Function_Declaration :
                      TYPE_FUNCTION id '(')' '{'
                      {
                        assert_no_redeclared_var($2,PL_FUNCTION);
                        printf("\t\t\t\t\t\t//-+++-Function-Declaration-Start-+++\n");
                        insert_function($2);
                        printf("\t\tjump-endfunction%s\n",$2);
                        printf("startfunction%s:\n",$2);
                        printf("\t\tnop\t\t//-no-operation\n");
                      }
                      Instructions Return_Statement
                      '}'
                      {
                        printf("\t\tstoreg-%d\t//-store-returned-value-of--%s\n",global_pos($2),$2);
                        printf("\t\treturn\n");
                        printf("endfunction%s:\n",$2);
                        printf("\t\t\t\t\t\t//-----Function-Declaration-End----\n");
                      }
                      ;

Function_Invocation :
                     PL_CALL id '(' ')'
                     {
                       assert_declared_var($2,PL_FUNCTION);
                       printf("\t\tpusha-startfunction%s\n",$2);
                       printf("\t\tcall\n");
                       printf("\t\tpushg-%d\t//-pushes-returned-value-of--%s\n",global_pos($2),$2);
                     }
                     ;
```

Denote que foi adicionada instrução **NOP** após a label do subprograma dado que a máquina virtual "saltava"a instrução seguinte à label sem a correr. Desta forma garantimos a correcta implementação da funcionalidade.

Ora, o código C adicionado às produções recorre a 2 métodos auxiliares ainda não definidos. De seguida apresentam-se os mesmos:

4.2.3.1 Método auxiliar: void insert_function (char* function_name);

```
void insert_function ( char* function_name ) {
    var_table[var_index].varname = strdup( function_name );
    var_table[var_index].value = -1;
    var_table[var_index].type = PL_FUNCTION;
    var_table[var_index].rows = -1;
    var_table[var_index].cols = -1;
    var_table[var_index].size =-1;
    int old_size = ia[var_index];
    var_index++;
    ia[var_index] = old_size + 1;
    printf("\t\tpushi-0\t\t//-space-for-function-%s-returned-value\n", function_name);
```

```
12 }
```

4.2.3.2 Método auxiliar: void assert_declared_var(char* varname, var_type type);

```
1 void assert_declared_var(char* varname, var_type type){
2     if ( !exists_var(varname, type) ){
3         compile_error("accessing non declared VAR");
4     }
5 }
```

4.2.4 Atribuição de valores a variáveis

Sempre que são reconhecidas as produções de atribuição de valores a variáveis é necessário incluir os código máquina responsáveis por tais operações. Ora, a não terminal **Assignement** dá origem a duas produções, sendo estas a atribuição de valores a variáveis do tipo inteiro atómico, e a atribuição de variáveis do tipo array ou matriz.

Denote que o não terminal **Assignement_Value** dá origem a três produções, cada uma representando uma atribuição de "origem"distinta. Pela análise das produções, podemos retirar que as atribuições poderão ser de valores lidos do standard input, de valores retornados por funções ou do resultado de expressões aritméticas.

O conjunto de instruções de código máquina que permitirão a correcta atribuição de valores a variáveis será incluído nas produções:

```
1 Assignment : id '=' Assignement_Value
2            | Vectors
3            '=' Assignement_Value
4            ;
5
6 Assignement_Value : Arithmetic_Expression
7                   | Read_Stdin
8                   | Function_Invocation
9                   ;
```

Sendo o seguinte código associado à produção:

```
1 Assignment : id '=' Assignement_Value
2            {
3              assert_declared_var($1, PL_INTEGER );
4              printf("\t\tstoreg %d\t// store var %s\n",global_pos($1),$1);
5            }
6            | Vectors
7            '=' Assignement_Value
8            {
9              printf("\t\tstoren\n");
10           }
11           ;
12
13 Assignement_Value : Arithmetic_Expression
14                   | Read_Stdin
15                   | Function_Invocation
16                   ;
```

Aprofundemos a nossa análise aos não terminais **Arithmetic_Expression** e **Read_Stdin**, dado que já analisamos anteriormente o não terminal **Function_Invocation**.

4.2.4.1 Análise às produções do não terminal Arithmetic_Expression

Sempre que são reconhecidas as produções que reflectem expressões aritméticas é necessário incluir os código máquina responsáveis por tais operações.

Desta forma, o conjunto de instruções de código máquina que permitirão a correcta atribuição de valores a variáveis será incluído nas produções:

```
Arithmetic_Expression : Term
                      | Arithmetic_Expression '+' Term
                      | Arithmetic_Expression '-' Term
                      ;

Term     : Factor
         | Term '*' Factor
         | Term '/' Factor
         | Term '%' Factor
         ;

Factor   : num
         | id
         | Vectors
         | '(' Arithmetic_Expression ')'
         ;
```

Sendo o seguinte código associado à produção:

```
Arithmetic_Expression : Term
                      | Arithmetic_Expression '+' Term { printf("\t\tadd\n");}
                      | Arithmetic_Expression '-' Term { printf("\t\tsub\n");}
                      ;

Term     : Factor
         | Term '*' Factor  { printf("\t\tmul\n");}
         | Term '/' Factor  { printf("\t\tdiv\n");}
         | Term '%' Factor  { printf("\t\tmod\n");}
         ;

Factor   : num
         { printf("\t\tpushi %d\n", $1); }
         | id
         {
           assert_declared_var($1, PL_INTEGER);
           printf("\t\tpushg %d\n", global_pos($1));
         }
         | Vectors { printf("\t\tloadn \n");}
         | '(' Arithmetic_Expression ')'
         ;
```

Como pode constatar pela produção **Factor : Vectors ;** é necessário analisarmos também o não terminal Vectors dado que é o responsável por gerar código máquina para aceder a valores do tipo array ou matriz. Analisemos de seguida esse não terminal.

4.2.4.2 Análise às produções do não terminal Vectors

Sempre que são reconhecidas as produções que reflectem a utilização de variáveis do tipo array ou matriz é necessário incluir os código máquina responsáveis por aceder aos valores das mesmas.

Antes de procedermos à especificação do acesso às posições do gp[n] que contêm os dados relativos às matrizes e arrays, devemos explicitar a forma com esses mesmos dados são armazenados.

Considere o exemplo de declaração de uma matriz de tamanho 2x4:

```
// declaracao de variavel matriz
int matriz_exemplo[2,4];

//associacao de valores inteiros aos indices da matrix
matriz_exemplo[0,1] = 5;
matriz_exemplo[1,3] = 10;
```

O dados da matriz são declarados de forma "row-wise"(linha a linha), sendo que, quando pretendemos aceder por exemplo ao elemento localizado na 2ª linha 4ª coluna, com "zero indexing", estaremos a aceder à posição matriz_exemplo[1,3], sendo que a posição de memória correspondente é dada por 1(linha do elemento) * nº de colunas +

3 (coluna do elemento).
Por forma a facilitar a compreensão da localização relativa ao gp, o mesmo exemplo foi transformado em ilustrações:

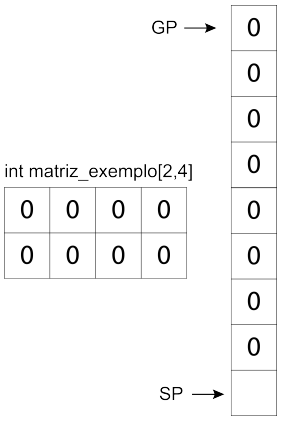

Figura 4.1: Declaração da variável do tipo matriz de tamanho 2x4 pela linha de código : int matriz_exemplo[2,4];

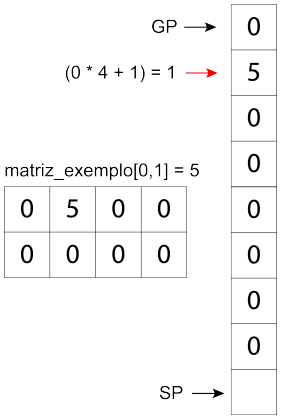

Figura 4.2: Atribuição do valor 5 à posição da matriz (1ª linha, 2ª coluna) pela linha de código : matriz_exemplo[0,1] = 5;

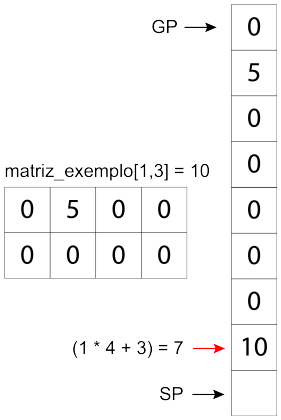

Figura 4.3: Atribuição do valor 10 à posição da matriz (2ª linha, 4ª coluna) pela linha de código : matriz_exemplo[1,3] = 10;

O acesso à memória pode ser descrito pelo seguinte esquema:

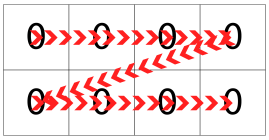

Figura 4.4: Ilustração do acesso à memória – "row wise"

Agora com este conhecimento relativo à forma como os dados estão posicionados podemos analisar o conjuto de produções responsáveis por traduzir esta forma de acesso em instruções de código máquina:

```
Vectors : id
        '['
        Arithmetic_Expression
        Second_Dimension  Dimension_End
        ;

Second_Dimension : ','  Arithmetic_Expression
                 | /*empty*/
                 ;

Dimension_End : ']'
              ;
```

Sendo o seguinte código associado à produção:

```
Vectors : id
        {
        if ( is_vector($1) ){
          assert_declared_var($1,PL_ARRAY);
        }
        else{
          assert_declared_var($1,PL_MATRIX);
        }
          printf("\t\tpushgp\n");
          printf("\t\tpushi %d\t//puts on stack the address of %s\n",global_pos($1),$1 );
          printf("\t\tpadd\n");
          printf("\t\t\t\t\t\t// +++ Matrix or Vector Dimension Start +++\n");
        }
        '['
        Arithmetic_Expression
        {
        if ( is_vector($1) ){
        }
        else {
          printf("\t\tpushi %d\t\t\t\t//pushes column size of vector or matrix\n",get_matrix_ncols(
    $1));
          printf("\t\tmul\n");
        }
        }
        Second_Dimension  Dimension_End
        ;

```

```
Second_Dimension : ','  Arithmetic_Expression
                 | /*empty*/ {printf("\t\tpushi 0\t\t//second dimension size of vector(0)\n");}
                 ;

Dimension_End : ']'
              {
                printf("\t\tadd\t\t//sums both dimensions\n");
                printf("\t\t\t\t\t\t// ---- Matrix or Vector Dimension End ---\n");
              }
              ;
```

Ora, o código C adicionado às produções recorre a 2 métodos auxiliares ainda não definidos. De seguida apresentam-se os mesmos:

4.2.4.3 Método auxiliar: int global_pos(char* varname);

```
int global_pos(char* varname) {
    int i, result;
    i = 0;
    while( (i < var_index ) && (strcmp(var_table[i].varname, varname)!= 0)){ i++; }
    if ( i == var_index ) {
        result = -1;
    }
    else{
        result = ia[i];
    }
    return result;
}
```

4.2.4.4 Método auxiliar: int is_vector(char* varname);

```
int is_vector(char* varname) {
    int i, r;
    i = 0;
    while (( i < var_index ) && (strcmp(var_table[i].varname, varname)!= 0)) i++;
    if ( i == var_index ) {
        r = 0;
    }
    else {
        if( var_table[i].type == PL_ARRAY ){
            r = 1;
        }
        else{
            r = 0;
        }
    }
    return r;
}
```

4.2.4.5 Método auxiliar: int get_matrix_ncols(char* varname);

```
int get_matrix_ncols(char* varname){
    int i, result;
    i = 0;
    while( (i < var_index ) && (strcmp(var_table[i].varname, varname)!= 0)){ i++; }
    if ( i == var_index ) {
        result = -1;
    }
    else{
        result = var_table[i].cols;
    }
```

```
11      return result;
12  }
```

Podemos, em jeito de validação confirmar que o comportamento descrito é mesmo o verificado na execução da máquina virtual:

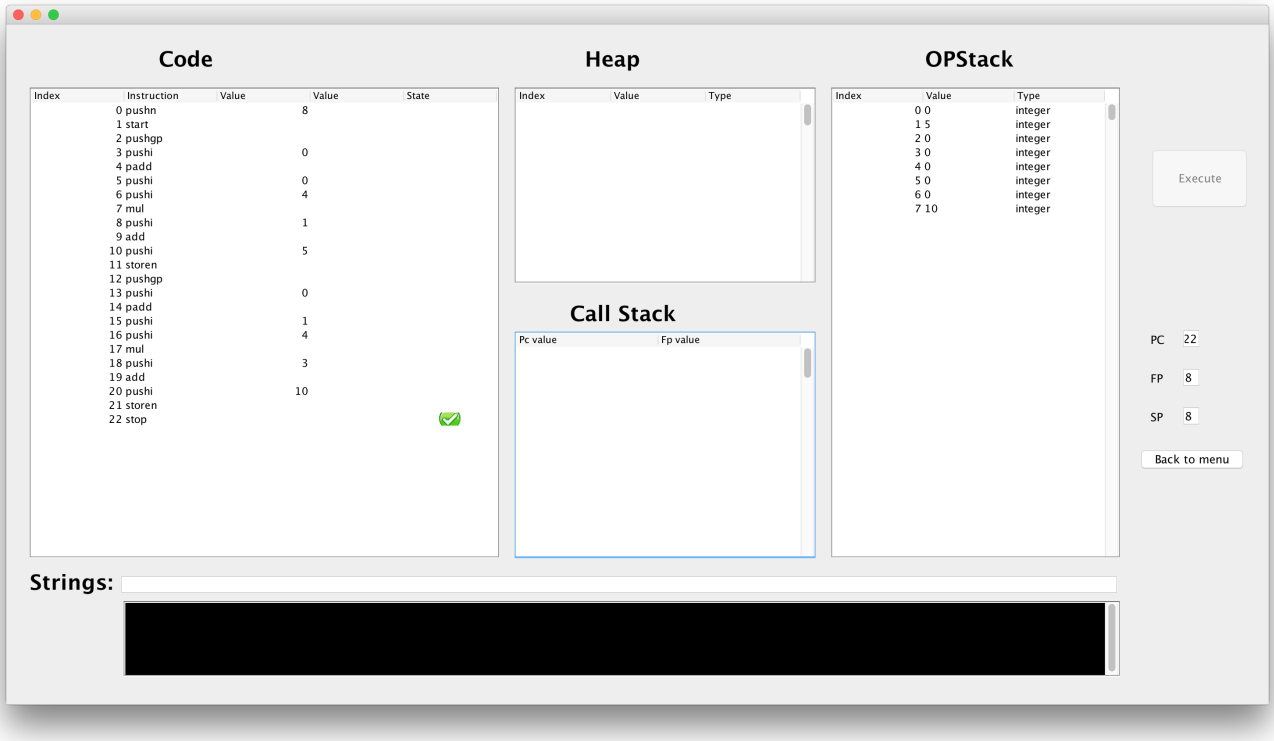

Figura 4.5: Confirmação do modo de registo em stack das variáveis do tipo matriz – "row wise"

4.2.4.6 Análise às produções do não terminal Read_Stdin

Sempre que são reconhecidas as produções que reflectem leitura do standard input é necessário incluir os código máquina responsáveis por tal operação.

Tal como foi descrito anteriormente, a leitura do standard input é apenas permitida aquando da associação do valor lido a uma variável.

O conjunto de instruções de código máquina que permitirão a correcta atribuição de valores a variáveis provenientes do standard input será dado pelas seguintes produções:

```
1 Read_Stdin : PL_READ '(' ')'
2            ;
```

Sendo o seguinte código associado à produção:

```
1 Read_Stdin : PL_READ '(' ')'
2            {
3              printf("\t\tread\n");
4              printf("\t\tatoi\n");
5            }
6            ;
```

4.2.5 Análise às produções do não terminal Logical_Expression

Sempre que são reconhecidas as produções que reflectem expressões lógicas é necessário incluir os código máquina responsáveis por tais operações.

As expressões lógicas e relacionais serão de extrema importância nas produções que envolvam condicionais e ciclos. Debruçar-nos-emos sobre essas instruções nas seções seguintes deste relatório.

Desta forma, o conjunto de instruções de código máquina que permitirão a correcta produção de valores lógicos 1 (verdade) e 0 (falso) serão incluídas nas produções:

```
Logical_Expressions : Logical_Expressions Logical_Expression
                    |
                    ;

Logical_Expression : '!' Relational_Expression
                   | Relational_Expression
                   | Logical_Expression '|''|' Relational_Expression
                   | Logical_Expression '&''&' Relational_Expression
                   ;
```

Sendo o seguinte código associado à produção:

```
Logical_Expressions : Logical_Expressions Logical_Expression
                    |
                    ;

Logical_Expression : '!' Relational_Expression
                   {
                     printf("\t\t\t\t\t\t// +++ Logical NOT BEGIN +++\n");
                     printf("\t\tpushi 1\n");
                     printf("\t\tadd\n");
                     printf("\t\tpushi 2\n");
                     printf("\t\tmod\n");
                     printf("\t\t\t\t\t\t// --- Logical NOT END ---\n");
                   }
                   | Relational_Expression
                   | Logical_Expression '|''|' Relational_Expression
                   {
                     printf("\t\t\t\t\t\t// +++ Logical OR BEGIN +++\n");
                     printf("\t\tadd\n");
                     printf("\t\tpushi 2\n");
                     printf("\t\tmod\n");
                     printf("\t\t\t\t\t\t// --- Logical OR END ---\n");
                   }
                   | Logical_Expression '&''&' Relational_Expression
                   {
                     printf("\t\t\t\t\t\t// +++ Logical AND BEGIN +++\n");
                     printf("\t\tmul\n");
                     printf("\t\tpushi 2\n");
                     printf("\t\tmod\n");
                     printf("\t\t\t\t\t\t// --- Logical AND END ---\n");
                   }
                   ;
```

Denote que tal como havia sido descrito anteriormente foi necessário proceder à inclusão de instruções de código máquina que conjugadas tenham o comportamento lógico da negação, OR e AND.

Nas produções aqui apresentados é introduzido um outro não terminal – **Relational_Expression**. Iremos analisar as produções com este relacionadas de seguida.

4.2.6 Análise às produções do não terminal Relational_Expression

Sempre que são reconhecidas as produções que reflectem expressões relacionais é necessário incluir os código máquina responsáveis por tais operações.

Desta forma, o conjunto de instruções de código máquina que permitirão a correcta produção de valores lógicos 1 (verdade) e 0 (falso) com base em relações serão incluídas nas produções:

```
Relational_Expression : Arithmetic_Expression
                      | Arithmetic_Expression '=''=' Arithmetic_Expression
                      | Arithmetic_Expression '!''=' Arithmetic_Expression
                      | Arithmetic_Expression '>' Arithmetic_Expression
                      | Arithmetic_Expression '>''=' Arithmetic_Expression
                      | Arithmetic_Expression '<' Arithmetic_Expression
                      | Arithmetic_Expression '<''=' Arithmetic_Expression
                      | '(' Logical_Expressions ')'
                      ;
```

Sendo o seguinte código associado à produção:

```
Relational_Expression : Arithmetic_Expression
                      | Arithmetic_Expression '=''=' Arithmetic_Expression
                      {
                        printf("\t\tequal\t//relational equal\n");
                      }
                      | Arithmetic_Expression '!''=' Arithmetic_Expression
                      {
                        printf("\t\t\t\t\t\t// +++ Logical NOT EQUAL BEGIN +++\n");
                        printf("\t\tequal\n");
                        printf("\t\tpushi 1\n");
                        printf("\t\tadd\n");
                        printf("\t\tpushi 2\n");
                        printf("\t\tmod\n");
                        printf("\t\t\t\t\t\t// --- Logical NOT EQUAL END ---\n");
                      }
                      | Arithmetic_Expression '>' Arithmetic_Expression
                      {
                        printf("\t\tsup\t//relational superior\n");
                      }
                      | Arithmetic_Expression '>''=' Arithmetic_Expression
                      {
                        printf("\t\tsupeq\t//relational superior or equal\n");
                      }
                      | Arithmetic_Expression '<' Arithmetic_Expression
                      {
                        printf("\t\tinf\t//relational inferior\n");
                      }
                      | Arithmetic_Expression '<''=' Arithmetic_Expression
                      {
                        printf("\t\tinfeq\t//relational inferior or equal\n");
                      }
                      | '(' Logical_Expressions ')'
                      ;
```

No seguimento do sucedido na seção 4.2.5 foi também para este tipo de produções necessário proceder à inclusão de instruções de código máquina que conjugadas tenham o comportamento lógico da negação de igualdade. Para os restantes operadores relacionais existiam instruções relacionais directamente presentes na linguagem máquina.

Capítulo 5

Testes às funcionalidades da Algebra

// cycle functions

5.0.0.1 Método auxiliar: int open_cycle();

5.0.0.2 Método auxiliar: int close_cycle();

// conditional functions

5.0.0.3 Método auxiliar: int open_conditional();

5.0.0.4 Método auxiliar: int close_conditional();

5.0.0.5 Método auxiliar: int current_conditional();

// error checking / handling functions

5.0.0.6 Método auxiliar: int exists_var(char* varname, var_type type);

Capítulo 6

Conclusão

Relativamente ao estado final do projecto acredito que foram cumpridos todos os requisitos, sendo que o segundo exercício foi sem dúvida o mais desafiante dada a enorme quantidade de dados e o tipo de dados em si a serem analisados. Reconhecer por si só quais as sequências de caracteres válidas foi um desafio.

Naturalmente que a partir da alínea 2.2.b a alínea 2.2.c foi de extrema facilidade, uma vez que todo o trabalho de análise já estava realizado.

Foi ainda tido em conta a possiblidade de recuperar de erros de leitura na alínea 2.2.b o que facilitou o input correct de dados e posterior tratamento. O recurso à biblioteca Glib, recomendada pelo professor José João num aula laboratorial permitiu-me ambientar ainda mais com código desenvolvido por terceiros e sua correcta análise e integração nos meus projectos.

Faço um balanço positivo do trabalho prático, pois, apesar de ser extremamente "time consuming"retirei muito conhecimento no que da análise de dados e processamento de linguagens diz respeito.

Apêndice A

Código do Programa da alínea 1a

Apêndice B

Código do Programa da alínea 2a

Apêndice C

Código do Programa da alínea 2b

Apêndice D

Código do Programa da alínea 3a