Uma Introdução ao GAP

(Notas de um mini-curso)

Manuel Delgado

Faculdade de Ciências da Universidade do Porto
http://www.fc.up.pt/cmup/home/mdelgado

Versão de 27 de Setembro de 2004

Conteúdo

Introdução

Estas notas foram escritas para servir de apoio a um mini-curso intitulado "Uma Introdução ao GAP" que apresentei, em Fevereiro de 2001, no Centro de Matemática da Universidade do Porto e no Centro de Álgebra (na Universidade de Lisboa). A menos desta introdução e de alguns pequenos detalhes, trata-se das notas distribuídas no início do curso.

Pretendi neste curso relatar um pouco da minha experiência pessoal com o GAP. Como acontece com muitas coisas ligadas aos computadores, não há como mexer nelas para perceber como funcionam. Nesse sentido, as notas começam com um capítulo verdadeiramente introdutório e que tem como objectivo fazer com que o leitor não deixe de experimentar por ter dificuldades, por exemplo, na instalação do GAP no seu computador.

Embora a partir daí os exemplos possam em geral parecer muito especializados, nenhuma parte do texto foi escrita a pensar apenas em especialistas de alguma área específica.

No segundo capítulo falo de autómatos... Porquê "autómatos"? O leitor não deve esquecer que tenho como objectivo relatar a minha experiência pessoal. Pretendo na segunda secção ilustrar a criação de um objecto novo. Fazendo alterações mínimas, poderia dizer que estava a falar de qualquer outra coisa. Deve sentir-se encorajado a criar outros objectos do seu interesse e que ainda não façam parte do GAP. Depois falo de funções que fazem parte de uma *share-package* em desenvolvimento e que se destina essencialmente a lidar com autómatos. Pretendo ilustrar a utilização de objectos recém criados.

No terceiro capítulo falo de funções que fazem parte de uma outra *share-package* em desenvolvimento e que se destina a lidar com semigrupos finitos. Os semigrupos são objectos que já fazem parte do GAP. Este capítulo é aproveitado para ilustrar alguns aspectos úteis do GAP.

Por fim, os inevitáveis exercícios.

Agradecimentos:

Aos Centro de Matemática da Universidade do Porto (CMUP) e Centro de Álgebra (CAUL) agradeço a oportunidade que me deram de levar a cabo a realização deste mini-curso;

Ao Vítor H. Fernandes agradeço o ter-me encorajado. A ele, e também à Helena Sezinando, agradeço a organização no CAUL.

Agradeço ainda o financiamento, parcial, da FCT através do *Centro de Matemática da Universidade do Porto* e também do Projecto SAPIENS 32817/99.

Capítulo 1

Primeiros passos

O GAP (**G**roups, **A**lgorithms and **P**rograming), começou por ser um sistema computacional para lidar com grupos, mas tem sido estendido em várias outras direcções. Basta olhar para o índice do manual e para o grande número de pacotes (*share-packages*) que têm surgido, para nos apercebermos disso.

O GAP tem sido utilizado na realização de muitos trabalhos de investigação como o atesta a grande quantidade de citações em artigos científicos (ver http://www.gap-system.org/Doc/Bib/bib.html). Tem também sido usado no ensino, especialmente para ensinar Teoria de Grupos.

Devo acrescentar que o GAP é gratuito (distribuído com as regras habituais para evitar o seu uso comercial). Começou a ser desenvolvido em 1984 em Aachen, na Alemanha, tendo o seu centro de desenvolvimento sido transferido para St. Andrews, na Escócia, em 1987. Há pessoas a trabalhar no GAP um pouco por todo o mundo.

Estas notas, escritas na primeira pessoa, baseiam-se na minha experiência pessoal com o GAP. Devo advertir o leitor de que, embora o GAP corra nos vários sistemas operativos (UNIX, MS-DOS, MacOS, ...), eu uso habitualmente o Linux, e poderá, nalguns casos, ser necessário fazer adaptações às quais não me refiro. Além dos manuais do GAP [GAP], foram de grande utilidade para a elaboração destas notas os trabalhos [BL, dG, Hu].

1.1 Como começ ar

A página *web* do GAP que aqui referirei várias vezes tem o endereço

```
http://www-gap.dcs.st-and.ac.uk/~gap/
```

O GAP consiste de um núcleo, uma livraria, bases de dados, *share-packages* e documentação. O núcleo está escrito em C e contém (entre outras coisas) um interpretador da linguagem GAP e, por questões de eficiência, as rotinas mais frequentemente utilizadas. A livraria está escrita em GAP e contém a quase totalidade dos algoritmos. Entre as bases de dados, existe, por exemplo, uma de grupos de pequena ordem. Existem, ou estão em preparação, *share-packages* que contêm algoritmos envolvendo grupos policíclicos, grupos automáticos ou bases de Gröbner, por exemplo.

A linguagem de programação GAP (tipo PASCAL) é de muito alto nível (está próxima da linguagem que nós falamos). O C, embora do mesmo tipo, é de mais baixo nível (está mais próximo da linguagem-máquina).

A instalação do GAP pode, em princípio, ser feita em qualquer plataforma, sendo também independente do sistema operativo usado. Num sistema Linux, em parte porque um sistema Linux contém sempre um compilador de C, a instalação é essencialmente trivial. Podem obter-se, a partir da página do GAP, núcleos compilados (i.e., já traduzidos para a linguagem máquina) para outros sistemas operativos.

A instalação completa do GAP (gap4r2), sem *share-packages*, ocupa presentemente cerca de 150 Mb de memória, mas podem fazer-se instalações que ocupam bastante menos espaço.

Para fazer a instalação deve começar por obter o ficheiro gap4r2.zoo (ou o basic4r2.zoo, se quiser instalar apenas o essencial e ocupar assim menos espaço) e, caso não o tenha feito ainda, o ficheiro unzoo.c. Estes ficheiros podem ser obtidos sem dificuldade a partir da página do GAP. (O leitor não deve ficar surpreendido se não encontrar o ficheiro gap4r2.zoo, mas o gap4r3.zoo ou outro posterior, em seu lugar.) Vou supor que a instalação vai ser feita no directório /caminho que contém os ficheiros unzoo.c e gap4r2.zoo e onde desde já nos encontramos (i.e., /caminho é o directório corrente). O unzoo.c precisa de ser compilado, o que pode ser feito executando

```
caminho> cc -o unzoo -DSYS_IS_UNIX -O unzoo.c
```

Para descompactar o gap4r2 deve fazer

```
caminho>  ./unzoo -x gap4r2.zoo
gap4r2/doc/aboutgap.tex -- extracted as text
... (muitas linhas parecidas com esta)
caminho>
```

Depois deve fazer

```
caminho> cd gap4r2
caminho/gap4r2> ./configure
checking host system type... i686-unknown-linux2.0.27
... (muitas linhas mais)
```

Depois

```
caminho/gap4r2> make
```

Fazendo agora

```
caminho/gap4r2> bin/gap.sh
...
gap>
```

inicia uma sessão GAP.

Antes de dar a instalação por completa, deve proceder à instalação de eventuais *bugfixs*, seguindo as instruções.

O manual

Uma instalação completa do GAP inclui diversas versões do manual (TEX, DVI, PDF, HTML) e também o manual *on-line*. (Dizer "dos manuais" seria mais correcto: existem de facto dois *tutorials* e dois manuais, a pensar no utilizador e no programador.) A impressão do manual é de todo desaconselhada: o manual do utilizador (do gap4r2) tem mais de 800 páginas. Considero a versão HTML particularmente útil. Para a consultar basta escrever no seu *browser*

```
file:/caminho/gap4r2/doc/htm/index.htm
```

Se estiver a usar a versão HTML do manual pode fazer *copy/paste* para testar os exemplos lá apresentados. O manual *on-line* é particularmente útil quando não se está na presença de um terminal gráfico.

1.2 Exemplos

Os exemplos desta secção destinam-se a familiarizar um pouco o leitor com a sintaxe do GAP. Não pretendo com eles ilustrar o que se pode fazer com o GAP, pois aquilo que eles cobrem é absolutamente insignificante.

Depois de ter o GAP devidamente instalado no seu computador, poderá começar uma sessão escrevendo, no *prompt* do seu terminal, a palavra gap seguida de um toque na tecla *return*. Quando se entra num programa é importante saber como sair dele, por isso devo desde já dizer que para terminar a sessão GAP bastará escrever quit; seguido de *return* ou, alternativamente, pressionado em simultâneo as teclas *ctrl-d*.

```
prompt> gap
...
gap> quit;
prompt>
```

A utilização do manual *on-line* pode ser feita do seguinte modo

```
gap> ?group
... # muita informação
```

Se preferir visualizar a ajuda no *Netscape* (que deve ter previamente aberto) pode fazer

```
gap> SetHelpViewer("Netscape");
The Help function will use netscape
gap> ?group
... # a informação é mostrada no Netscape
```

(Os pedidos de ajuda subsequentes recorrem ao *Netscape*. Se quiser mudar pode então escrever SetHelpViewer("Screen");. Esta mudança pode ser útil quando se pretendem usar *share-packages* em fase de desenvolvimento e que, já tendo alguma documentação, não a têm em HTML.)

As instruções GAP terminam sempre com ; (ponto e vírgula) e o GAP dá sempre uma resposta (que por vezes significa muito pouco). Quando não queremos ver a resposta escrevemos ;; em vez de ;. Os exemplos seguintes quase não precisam de comentários.

```
gap> s6 := Group( (1,2), (1,2,3,4,5,6) );;
gap> s8 := Group( (1,2), (1,2,3,4,5,6,7,8) );
Group([ (1,2), (1,2,3,4,5,6,7,8) ])
gap> Size(s8);
40320
gap> Factorial(8);
40320
gap> IsAbelian(s8);
false
gap> FactorsInt(40320);
[ 2, 2, 2, 2, 2, 2, 2, 3, 3, 5, 7 ]
gap> PrintFactorsInt(40320); Print( "\n" );
2^7*3^2*5*7
```

Basta olhar para a factorização de 40320 para sabermos que os 3-subgrupos de Sylow do grupo simétrico S_8 têm ordem 9. Assim, mesmo sem entrar em detalhes sobre o resultado produzido pela função SylowSubgroup, não ficamos surpreendidos com o resultado seguinte:

```
gap> Size(SylowSubgroup(s8,3));
9
```

Os nomes das funções que fazem parte do GAP começam sempre por maiúsculas e são em geral muito compridos. A edição tem muitas semelhanças com a edição de comandos em UNIX ou em EMACS. Considero particularmente útil o facto de a técnica de pressionar a tecla *tab* para completar os nomes dos comandos também funcionar aqui

```
gap> SylowS
```

Pressionando agora *tab* obtém-se

```
    SylowSubgroup
    SylowSubgroupOp
    SylowSubgroupPermGroup
    SylowSystem
gap> SylowS
```

O GAP é também uma linguagem de programação. Para evitar dar nomes que fazem parte do GAP às suas funções, basta começá-los por letras minúsculas. Pode escrever do modo seguinte um programa que lhe permite calcular o factorial de um número natural.

```
gap> fac:=function(n)
> local i, f;
> f:=1;
> for i in [1 .. n] do
> f:=f*i;
> od;
> return f;
> end;
function( n ) ... end
gap> fac(10);
3628800
gap> fac(8);
40320
```

Os programas assim escritos, perdem-se quando se termina a sessão. Como evitar isso? Uma maneira de guardar parte de numa sessão num ficheiro (com endereço trabalho/ficheiro) é fazer

```
gap> LogTo("trabalho/ficheiro");
... # tudo isto é guardado
gap> LogTo();
```

ficando então guardado em ficheiro tudo o que se passar entre os dois LogTo. Outra maneira é entrar num editor de texto dentro de uma sessão GAP (basta usar a instrução Edit(“ficheiro”), com a qual entra, por defeito, no *vi*), mas eu não o costumo fazer. O que eu costumo fazer é escrever os meus programas fora da sessão GAP (geralmente em paralelo) usando um editor de texto qualquer (de facto sempre o mesmo). Se pretender escrever os seus programas usando o *emacs*, o meu editor de texto favorito, chamo a sua atenção para o facto de existir o *gap-mode* que pode obter a partir da página *web* do GAP e que considero muito útil. A função fac poderia escrevê-la com o seguinte aspecto (note a *indentação* feita automaticamente no *gap-mode*)

```
fac:=function(n)
    local i, f;
    f := 1;
    for i in [1 .. n] do
        f := f*i;
    od;
    return f;
end;
```

no ficheiro fac.g. Quando quisesse depois utilizar esta função (e outras escritas no mesmo ficheiro), bastaria escrever Read(”caminho/fac.g”);. Não precisa de escrever o caminho se fac.g estiver no directório corrente.

```
gap> Read("caminho/fac.g");
gap> fac(13);
6227020800
```

O exemplo seguinte envolve o monóide dado pela apresentação: $B_2^1 = \langle a, b | a^2 = b^2 = 0, aba = a, bab = b \rangle$. (É escusado dizer que costumo guardar os exemplos mais complexos que uso com frequência num ficheiro o qual é lido usando a função Read de que já falei atrás. Ver também o ficheiro .gaprc na página 8.)

```
gap> f := FreeMonoid("a","b");
<free monoid on the generators [ a, b ]>
gap> a := GeneratorsOfMonoid( f )[ 1 ]; # tenho de dar nomes aos geradores
a
gap> b := GeneratorsOfMonoid( f )[ 2 ];;
gap> r:=[[a^3,a^2],
>       [a^2*b,a^2],
>       [b*a^2,a^2],
>       [b^2,a^2],
>       [a*b*a,a],
>       [b*a*b,b] ];
[ [ a^3, a^2 ], [ a^2*b, a^2 ], [ b*a^2, a^2 ], [ b^2, a^2 ], [ a*b*a, a ],
  [ b*a*b, b ] ]
gap> b21:= f/r;
<fp semigroup on the generators [ <identity ...>, a, b ]>
gap> GreensDClasses(b21);
[ {<identity ...>}, {a}, {a^2} ]
gap> Size(b21);
6
```

Se quisesse saber quanto tempo (de processador) tinha demorado esta última operação (o cálculo da ordem do monóide)

```
gap> time;
3310
```

Há outras maneiras de dar monóides

```
gap> g0:=Transformation([3,1,3]);
Transformation( [ 3, 1, 3 ] )
gap> g1:=Transformation([2,3,3]);
Transformation( [ 2, 3, 3 ] )
gap> poi2:= Monoid(g0,g1);
<monoid with 2 generators>
gap> g:=Transformation([2,1,3]);
Transformation( [ 2, 1, 3 ] )
gap> popi2:= Monoid(g,g1,g0);
<monoid with 3 generators>
gap> DisplayEggBoxOfDClass(GreensDClassOfElement(popi2,g0));
[ [  0,  1 ],
  [  1,  0 ] ]
```

```
gap> Size(popi2);
7
gap> time;
110
```

Outro exemplo:

```
gap> mat := [[18,5],[4,7],[2,12]];
[ [ 18, 5 ], [ 4, 7 ], [ 2, 12 ] ]
gap> HermiteNormalFormIntegerMat(mat);
[ [ 2, 0 ], [ 0, 1 ], [ 0, 0 ] ]
```

Em particular $\langle (2,0),(0,1)\rangle$ é uma base para o subgrupo de $\mathbb{Z}^2$ gerado pelo conjunto de vectores $\{(18,5),(4,7),(2,12)\}$.

O exemplo seguinte destina-se a chamar a atenção para as funções ShallowCopy (faz uma cópia do primeiro nível) e StructuralCopy (faz realmente uma cópia)

```
gap> lista := [1,2];
[ 1, 2 ]
gap> k := lista;
[ 1, 2 ]
gap> k[1] := 3;
3
```

Sabe o que se passou com lista;?

```
gap> lista;
[ 3, 2 ]
```

```
gap> registo := rec(vertices := [1,2,3],
> arestas :=[[1,2],[2,3]]);
rec( vertices := [ 1, 2, 3 ], arestas := [ [ 1, 2 ], [ 2, 3 ] ] )
gap> cp1 := ShallowCopy(registo);
rec( vertices := [ 1, 2, 3 ], arestas := [ [ 1, 2 ], [ 2, 3 ] ] )
gap> cp2 := StructuralCopy(registo);
rec( vertices := [ 1, 2, 3 ], arestas := [ [ 1, 2 ], [ 2, 3 ] ] )
gap> cp2.vertices;
[ 1, 2, 3 ]
gap> cp2.vertices[1] := 3;
3
gap> registo;
rec( vertices := [ 1, 2, 3 ], arestas := [ [ 1, 2 ], [ 2, 3 ] ] )
gap> cp1.arestas := [];; cp1.vertices[1] := 5;;
gap> cp1;
rec( vertices := [ 5, 2, 3 ], arestas := [  ] )
gap> registo;
rec( vertices := [ 5, 2, 3 ], arestas := [ [ 1, 2 ], [ 2, 3 ] ] )
```

1.3 *Share-packages*

As *share-packages* GAP são escritas num formato próprio o qual é análogo ao do próprio GAP. Contêm obrigatoriamente um directório doc com documentação, sendo o manual escrito em TEX. Um directório gap contendo os algoritmos e um ficheiro README bem como outros contendo informações sobre a versão, etc. são também obrigatórios.

Como uma forma de encorajar os trabalhos GAP, há presentemente a possibilidade de serem submetidos pacotes GAP os quais passam por um sistema de *referee* análogo ao de uma revista científica.

1.4 O ficheiro .gaprc

Quando numa sessão GAP se pretendem usar *share-packages*, deve usar-se a opção -l para indicar onde é que as *share-packages* estão colocadas. Tenho no meu ficheiro .bashrc o seguinte *alias*

```
alias gap4='/home/gap4r2/bin/gap.sh -l "/home/gap4r2;/home/mdelgado/aut-semi"'
```

que me permite carregar em todas as sessões os pacotes automata e finsemi, previamente colocados em aut-semi/pkg.

Ao iniciar uma sessão GAP o ficheiro .gaprc é dos primeiros a ser lido. Indico, a título de exemplo, o .gaprc que eu tenho presentemente.

```
#Change the variable EDITOR; works in gap3, but not in gap4
#EDITOR := "/usr/bin/emacs -nw";

#packages
RequirePackage("automata");
RequirePackage("finsemi");

#read examples
Read("/home/mdelgado/aut-semi/ex-finsemi");
Read("/home/mdelgado/aut-semi/ex-automata");

#abbreviations
rcg := RightCayleyGraph;
```

Capítulo 2

Autómatos e linguagens racionais

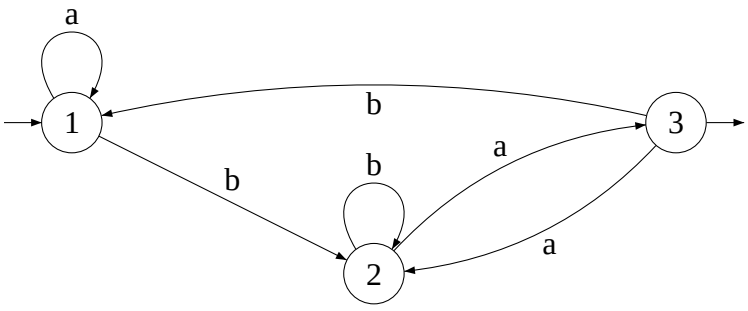
Num sistema como o GAP há sempre algo que seria desejável ter implementado mas ainda não está. Poderíamos gostar de ter disponíveis implementações de outros algoritmos para lidar com objectos já existentes (por exemplo semigrupos), ou mesmo ter disponíveis objectos novos. Neste capítulo vamos ver como criar um objecto que, para já, não faz parte do GAP: um autómato. Obviamente o leitor não está dispensado da consulta dos manuais se quiser criar os seus próprios objectos. Em particular, o *programmer's tutorial* contém exemplos de índole um pouco diferente do aqui apresentado. Pode também ser muito útil o trabalho [dG] de Willem de Graaf (a cuja página pessoal pode aceder via a página do GAP a qual contém também apontadores para páginas de diversos outros autores).

Este capítulo baseia-se numa versão muito preliminar daquilo que poderá vir a ser uma *share-package* sobre autómatos e à qual dou o nome *automata*. Esse pacote já conta com alguma documentação que poderá ser utilizada para completar certos detalhes.

Procurarei manter informação actualizada sobre esse trabalho no directório gap-aut-semi da minha página *web*.

2.1 Um pouco sobre autómatos

Um autómato pode muitas vezes ser descrito por um diagrama como o que se segue.



Este exemplo descreve um autómato com 3 estados (precisamente os elementos do conjunto $\{1,2,3\}$). A seta a apontar para o estado 1 indica que 1 é um estado inicial e a seta a sair do estado

3 indica que 3 é um estado final. O conjunto $\{a, b\}$ é o alfabeto do autómato (i.e., o conjunto das letras que podem ser etiqueta de alguma aresta). Trata-se de um autómato *determinístico*, já que não existem duas arestas etiquetadas pela mesma letra a sair de um estado. As palavras ba, ba^2a, a^5ba e bb^na, com $n \in \mathbb{N}$, são exemplos de palavras reconhecidas pelo autómato, já que são rótulos de caminhos que vão do estado inicial ao estado final. O conjunto das palavras nestas condições diz-se a *linguagem* do autómato.

O autómato do exemplo anterior (ou o grafo que lhe está subjacente) está intimamente relacionado com as transformações que as letras a e b induzem no conjunto dos vértices:

	1	2	3
a	1	3	2
b	2	2	1

Estas transformações geram um submonóide (dito *monóide de transições do autómato*) do monóide das transformações do conjunto dos estados do autómato em si próprio, com a operação de composição.

Existe uma extensa bibliografia onde pode encontrar muito sobre autómatos. Escolho [H&U] para lhe sugerir.

2.2 Como criar novos objectos: um exemplo

Nesta secção digo como criar um objecto chamado Automaton. O nome está para *autómato* (o qual pode ou não ser determinístico). Lembro que estes objectos correspondem ao estado actual de uma versão preliminar daquilo que poderá vir a ser uma *share-package* sobre autómatos.

Em GAP, um objecto começa por ser um "saco" capaz de conter informação. É criado usando a função Objectify. Atendendo a que para o uso comum do GAP não há qualquer necessidade de saber como os objectos são criados, não vou entrar em grandes detalhes, limitando-me a indicar uma maneira de criar um objecto que, se espera, possa comportar-se como um autómato.

As linhas que se seguem são praticamente retiradas da *share-package automata* onde estes objectos são criados. Começo por escolher uma representação (neste caso é um `registo` em que os campos são type, states, etc.).

```
DeclareRepresentation( "IsAutomatonRep", IsComponentObjectRep,
        ["type","states","alphabet","transitions","initial","accepting"]);
```

Depois construo a família de todos os autómatos e um tipo:

```
F:= NewFamily( "Automata" , IsAutomatonObj );
T := NewType( F, IsAutomatonObj and IsAutomatonRep );
```

Posso agora usar a função Objectify, o que vou fazer dentro da função (global) Automaton que indico a seguir (omitindo o que não me parece essencial nesta fase).

```
InstallGlobalFunction( Automaton, function(Type, Size, SizeAlphabet,
```

```
        TransitionTable, ListInitial, ListAccepting )

    local A, aut;

    #some tests...
    if not IsPosInt(Size) and IsPosInt(SizeAlphabet) and
       IsMatrix(TransitionTable) and IsList(ListInitial) and
       IsList(ListAccepting) then
       Error( "you must give the specifications of an automaton" );
    fi;

    aut := rec(type := Type,
               alphabet := SizeAlphabet,
               states := Size,
               initial := ListInitial,
               accepting := ListAccepting,
               transitions := TransitionTable );

    A := Objectify( T,
                 aut );

    # Return the automaton.
    return aut;
end);
```

Posso agora construir um autómato do modo seguinte

```
gap> aut:=Automaton("det",4,2,[[3,,3,4],[3,4,0,4]],[1],[4]);;
```

O modo como este objecto é impresso depende da função

```
#############################################################################
##
#M  PrintObj( <A> ) . . . . . . . . . . . . print automata
##
InstallMethod( PrintObj,
    "special for a deterministic automaton",
    true,
    [IsAutomatonObj and IsAutomatonRep], 0,
function( A )
    local au, aut, l1, l2, arr, array, i, j, letters, max, R, l, k, s, t;

    letters := [];
    aut := A;
    au := StructuralCopy(aut!.transitions);
```

```
for i in [1 .. Length(aut!.transitions)] do
    for j in [1 .. Length(aut!.transitions[1])] do
        if not IsBound(au[i][j]) or au[i][j] = 0 then
            au[i][j] := " ";
        fi;
    od;
od;

if aut!.alphabet < 5 then     ##  for small alphabets, the letters
                                  ##  a, b, c, d are used
    letters := ["a", "b", "c", "d"];
    else
    for i in [1 .. aut!.alphabet] do
        Add(letters, Concatenation("a", String(i)));
    od;
fi;
l2 := [];
array := [];
s := [];
arr := List( au, x -> List( x, String ) );
max := Maximum( List( arr, x -> Maximum( List(x,Length) ) ) );

if aut!.states < 16 then
    l1 := Concatenation(["" ,"|" ],[1 .. aut!.states]);
    for j in [1 .. max] do
        s := Concatenation(s,"-");
    od;

    for i in [1 .. aut!.states + 2] do
         l2[i] := s;
    od;
    for i in [1 .. aut!.alphabet + 2] do
        if i = 1 then
            array[i] := l1;
        elif i = 2 then
            array[i] := l2;
        else
            array[i] := Concatenation([letters[i-2],"|"], au[i-2]);
        fi;
    od;
else
    for i in [1 .. aut!.states] do
        for j in [1 .. aut!.alphabet] do
            if IsBound(au[j]) and IsBound(au[j][i]) and
```

```
                au[j][i] <> " " then
                  Add(array, [i, letters[j], au[j][i]]);
              fi;

          od;
      od;
  fi;


  arr := List( array, x -> List( x, String ) );
  max := Maximum( List( arr, x -> Maximum( List(x,Length) ) ) );

  Print( " " );
  for l  in [ 1 .. Length( arr ) ]  do
      if l > 1  then
          Print( " " );
      fi;
      Print( " " );
      for k  in [ 1 .. Length( arr[ l ] ) ]  do
          Print( String( arr[ l ][ k ], max + 1 ) );
          if k = Length( arr[ l ] )  then
              Print( " " );
          else
              Print( " " );
          fi;
      od;
      if l = Length( arr )  then
          Print( "\n" );
      else
          Print( "\n" );
      fi;
  od;
  Print("Initial state: ", aut!.initial,"\n");
  if Length(aut!.accepting) = 1 then
      Print("Accepting state: ", aut!.accepting,"\n");
  else
      Print("Accepting states: ", aut!.accepting,"\n");
  fi;
end);
```

Obtém-se agora o seguinte

```
gap> aut;
      |  1  2  3  4
   -  -  -  -  -  -
   a  |  3     3  4
   b  |  3  4     4
Initial state: [ 1 ]
Accepting state: [ 4 ]
```

Também podemos criar um autómato não determinístico

```
gap> ndaut:=Automaton("nondet",4,2,[[3,[1,2],3,0],[3,4,0,[2,3]]],[1],[4]);
                     |         1         2         3         4
   --------  --------  --------  --------  --------  --------
           a         |     [ 3 ]  [ 1, 2 ]     [ 3 ]     [ 0 ]
           b         |     [ 3 ]     [ 4 ]     [ 0 ]  [ 2, 3 ]
Initial state: [ 1 ]
Accepting state: [ 4 ]
```

A função PrintObj acima prevê que possamos querer usar autómatos com muitos estados

```
gap> aut:=Automaton("det",16,2,[[4,0,0,6,3,1,4,8,7,4,3,0,6,1,6,0],
> [3,4,0,0,6,1,0,6,1,6,1,6,6,4,8,7,4,5]],[1],[4]);
    1   a   4
    1   b   3
    2   b   4
      ...    # outras linhas
   15   b   8
   16   b   7
Initial state: [ 1 ]
Accepting state: [ 4 ]
```

2.3 Algumas funções da *share-package “automata”*

Nas secções anteriores já falei um pouco sobre autómatos e incluí alguns exemplos. Os exemplos sobre autómatos continuam aqui, na primeira subsecção, estando as outras duas subsecções dedicadas a conjuntos racionais e semilineares.

2.3.1 Autómatos

Antes de continuar devo referir que o pacote *automata* conta desde já com a colaboração de outras pessoas às quais devo grande parte do trabalho de programação das funções apresentadas nesta subsecção:

– Cyril Nicaud (Universidade de Paris 7) implementou, em GAP 3, um algoritmo devido a Hopcroft para reduzir um autómato determinístico (e que facilmente leva à produção de um autómato minimal);

– Paulo Varandas, num trabalho da disciplina de Teoria Algébrica dos Autómatos (da licenciatura em Matemática da Universidade do Porto), implementou, entre outros, um algoritmo para determinizar um autómato.

Além dos exemplos aqui apresentados (e que quase não precisam de comentários) estão já programadas algumas outras funções. Remeto o leitor para o manual do pacote *automata*.

```
gap> aut:=Automaton("det",4,2,[[3,3,3,4],[3,4,3,4]],[1],[4]);
      :  1  2  3  4
   -  -  -  -  -  -
   a  :  3  3  3  4
   b  :  3  4  3  4
Initial state: [ 1 ]
Accepting state: [ 4 ]

gap> MinimalizeDDAutomaton(aut);
      :  1
   -  -  -
   a  :  1
   b  :  1
Initial state: [ 1 ]
Accepting states: [  ]
```

O algoritmo usado para produzir um autómato determinístico reconhecendo a mesma linguagem que o original (não determinístico) é o da *construção dos subconjuntos*. É claro que uma vez produzido um autómato determinístico, ele pode ser minimalizado.

```
gap> aut:=Automaton("nondet",3,2,[[1,[2,3],1],[2,1,3]],[1],[2]);
                    |         1         2         3
   --------  --------  --------  --------  --------
          a         |     [ 1 ]  [ 2, 3 ]     [ 1 ]
          b         |     [ 2 ]     [ 1 ]     [ 3 ]
Initial state: [ 1 ]
Accepting state: [ 2 ]

gap> SubSetAutomaton(aut);
      |  1  2  3  4  5  6  7  8
   -  -  -  -  -  -  -  -  -  -
   a  |  1  2  4  4  2  7  4  2
   b  |  1  6  3  4  7  2  5  8
Initial state: [ 2 ]
Accepting states: [ 3, 4, 6, 7 ]
```

```
gap> aut:=Automaton("det",4,2,[[3,1,3,4],[3,4,2,4]],[1],[4]);;
gap> Size(TransitionSemigroup(aut));
24
```

2.3.2 Conjuntos racionais

Os subconjuntos de um monóide M que podem ser obtidos a partir dos subconjuntos singulares por meio de um número finito de uniões, produtos e "estrelas" (a estrela tem o significado de "submonóide gerado por") juntamente com o conjunto vazio, são os *subconjuntos racionais* de M. Podemos então exprimir qualquer subconjunto racional não vazio partindo de subconjuntos singulares e usando um número finito de vezes a união, o produto e a operação estrela. Uma tal expressão diz-se uma *expressão racional* desse subconjunto. Por exemplo, o conjunto $\{ba^n a : n \in \mathbb{N}\}$, que está contido na linguagem do autómato da página 9, pode ser representado pela expressão racional bb^*a. É claro que expressões racionais diferentes podem representar o mesmo conjunto.

O pacote automata também serve para trabalhar com expressões racionais de subconjuntos racionais do monóide livre, também ditos *linguagens racionais*.

A expressão racional (a(aUb))* (mostrada com este aspecto em virtude de uma função PrintObj) pode ser fornecida ao GAP do seguinte modo

```
gap> RatExpOnnLetters(2,"star",RatExpOnnLetters(2,"product",
> [RatExpOnnLetters(2,[],[1]),RatExpOnnLetters(2,"union",
> [RatExpOnnLetters(2,[],[1]),RatExpOnnLetters(2,[],[2])])]));
(a(aUb))*
```

Pode também ser obtida do modo indicado a seguir e que torna fácil a construção de expressões racionais mais complexas

```
gap> r1 := RatExpOnnLetters(2,[],[1]);
a
gap> r2 := RatExpOnnLetters(2,[],[2]);
b
gap> r3 := UnionRatExp(r1,r2);
aUb
gap> r4 := ProductRatExp(r1,r3);
a(aUb)
gap> r5 := StarRatExp(r4);
(a(aUb))*
```

Posso determinar um autómato que reconheça a linguagem dada por esta expressão racional

```
gap> RatExpToAutomaton(r5);
      |  1  2  3
   -  -  -  -  -
   a  |  1  3  2
   b  |  1  1  2
Initial state: [ 2 ]
Accepting state: [ 2 ]
```

e posso também determinar uma expressão racional para a linguagem reconhecida por um autómato

```
gap> A:=Automaton("det",10,2,[[7,5,7,5,4,9,10,9,10,9],
> [8,6,8,9,9,1,3,1,9,9]],[2],[6,7,8,9,10]);
        :   1   2   3   4   5   6   7   8   9  10
   --  --  --  --  --  --  --  --  --  --  --  --
    a   :   7   5   7   5   4   9  10   9  10   9
    b   :   8   6   8   9   9   1   3   1   9   9
Initial state: [ 2 ]
Accepting states: [ 6, 7, 8, 9, 10 ]

gap> AutomatonToRatExp(A);
b(b(aUb))*U(aa*bUb(b(aUb))*a)(aUb)*
```

A forma relativamente simplificada desta expressão deve-se ao facto de a função AutomatonToRatExp começar por minimalizar o autómato A.

2.3.3 Conjuntos semilineares

Os subconjuntos racionais do monóide comutativo livre $\mathbb{N}^n$ são os *subconjuntos semilineares*, isto é, uniões finitas de subconjuntos da forma $a+b_1\mathbb{N}+\cdots+b_p\mathbb{N}$, com $a,b_1,\ldots,b_p\in\mathbb{N}^n$ (aos quais se chama *lineares*). O fecho profinito (i.e. para a topologia profinita), em $\mathbb{Z}^n$, de $a+b_1\mathbb{N}+\cdots+b_p\mathbb{N}$ é $a+b_1\mathbb{Z}+\cdots+b_p\mathbb{Z}$. A uma união finita de subconjuntos desta forma (ditos $\mathbb{Z}$-*lineares*) chamo *conjunto* $\mathbb{Z}$-*semilinear*. (Note-se que os subconjuntos $\mathbb{Z}$-semilineares são uniões finitas de classes laterais de subconjuntos de $\mathbb{Z}^n$.)

O conjunto linear $(0,1)+(1,2)\mathbb{N}+(5,2)\mathbb{N}$ pode ser dado do seguinte modo

```
gap> LN := NLinear(2,[0,1],[[1,2],[5,2]]);
[ 0, 1 ] + [ 1, 2 ] N  + [ 5, 2 ] N
```

Se quiser dar o conjunto $\mathbb{Z}$-linear $(0,1)+(1,2)\mathbb{Z}+(5,2)\mathbb{Z}$, obtenho um resultado diferente:

```
gap> LZ := ZLinear(2,[0,1],[[1,2],[5,2]]);
[ 0, 1 ] + [ 1, 2 ] Z  + [ 0, 8 ] Z
```

Observo que a matriz $\begin{pmatrix}1&2\\0&8\end{pmatrix}$ é a forma normal hermítica da matriz $\begin{pmatrix}1&2\\5&2\end{pmatrix}$, logo o conjunto $\{(1,2),(0,8)\}$ gera precisamente o mesmo subgrupo de $\mathbb{Z}^n$ que $\{(1,2),(5,2)\}$, donde $(0,1)+(1,2)\mathbb{Z}+(5,2)\mathbb{Z}=(0,1)+(1,2)\mathbb{Z}+(0,8)\mathbb{Z}$.

No meu trabalho tenho tido várias vezes necessidade de calcular a imagem comutativa da linguagem de um autómato (a qual é um conjunto semilinear) bem como o seu fecho profinito (um conjunto $\mathbb{Z}$-semilinear).

```
 gap> aut := Automaton("det",5,2,[[1,2,4,2,1],[1,1,1,5,1]],[3],[2,3,4,5]);
      |  1  2  3  4  5
   -  -  -  -  -  -  -
   a  |  1  2  4  2  1
   b  |  1  1  1  5  1
Initial state: [ 3 ]
Accepting states: [ 2, 3, 4, 5 ]

gap> AutomatonToRatExp(aut);
1UabUa(1Uaa*)
gap> LanguageCom(aut); # imagem comutativa
[ [ 1, 1 ], [ 0, 0 ], [ 1, 0 ], [ 2, 0 ], [ 3, 0 ] + [ 1, 0 ] N  ]
gap> LanguageAb(aut);  # fecho da imagem comutativa
[ [ 1, 1 ], [ 0, 0 ] + [ 1, 0 ] Z  ]
```

Capítulo 3

Semigrupos finitos

Neste capítulo falo de uma *share-package* à qual presentemente dou o nome de *finsemi* e que, tal como a *share-package automata*, está em desenvolvimento. Ela contém desde já alguma documentação que pode ser usada para completar alguns detalhes omitidos nete capítulo e destina-se a lidar com semigrupos finitos. Espero não perder nesta fase os leitores menos habituados à linguagem dos semigrupos, pois a maior parte das coisas que vou dizer são perfeitamente gerais e o leitor pode facilmente imaginar exemplos na sua área preferida.

Para motivação e teoria sugiro a bibliografia [Al, D:98, D, DF].

3.1 Alguns monóides

Já no primeiro capítulo indiquei duas possibilidades para dar um monóide. Lembro que B_2^1 (ver página 6), dito o *monóide de Brandt com 6 elementos*, foi dado por meio de geradores e relações, sendo as relações $a^2 = b^2 = 0, aba = a, bab = b$, para os geradores a e b. O *grafo de Cayley (direito)* de um monóide M com geradores $a, b, \ldots$ é um grafo cujos vértices representam os elementos de M, havendo uma aresta de um vértice x para um vértice y etiquetada por um gerador g se $xg = y$. Não surpreende assim o seguinte:

```
gap> rcg := RightCayleyGraph(b21);
      |  1  2  3  4  5  6
   -  -  -  -  -  -  -  -
   a  |  2  4  6  4  2  4
   b  |  3  5  4  4  4  3
Initial state: [  ]
Accepting states: [  ]
```

(Os grafos de Cayley são vistos como autómatos sem estados iniciais nem finais.) O grafo de Cayley pode ser transformado num autómato por escolha de um estado final

```
gap> AutCayley(rcg,3);
      |  1  2  3  4  5  6
   -  -  -  -  -  -  -  -
   a  |  2  4  6  4  2  4
   b  |  3  5  4  4  4  3
Initial state: [ 1 ]
Accepting state: [ 3 ]
```

Mais dois monóides (de transformações parciais injectivas de um conjunto com 4 elementos; o número 5 aparece por razões técnicas: o GAP não trabalha com transformações parciais):

```
gap> g0:=Transformation([5,1,2,3,5]);;
gap> g1:=Transformation([1,2,4,5,5]);;
gap> g2:=Transformation([1,3,5,4,5]);;
gap> g3:=Transformation([2,5,3,4,5]);;
gap> g:=Transformation([2,3,4,1,5]);;
gap> poi4:= Monoid(g0,g1,g2,g3);;
gap> popi4:= Monoid(g,g0,g1,g2,g3);
gap> Size(poi4);Size(popi4);
70
141
```

3.2 Núcleos

Podemos usar o GAP para calcular o núcleo de um monóide (usando GroupKernel). É bem conhecido dos semigrupistas (peço aos que o não são para imaginarem outro exemplo) que há casos em que o núcleo de um monóide pode ser calculado de forma muito eficiente: quando os idempotentes comutam, por exemplo. Assim, para a operação GroupKernel podemos ter um método para calcular o núcleo de um monóide cujos idempotentes comutam (facto que pode eventualmente já ser conhecido do próprio monóide) e outros métodos mais gerais. O GAP escolhe depois o método mais adequado. Posso escrever a operação GroupKernel (a qual pretendo que seja um "atributo") com o seguinte aspecto

```
GroupKernel := NewAttribute("GroupKernel",IsSemigroup);
##
InstallMethod(GroupKernel,"for finite semigroups", true, [IsSemigroup], 1,
function(M)
    local nwcl;
    if not HasCommutingIdempotents(M) then
        TryNextMethod();
    fi;
    return Semigroup(GeneratorsOfIdempotents(M));
end);
```

```
InstallMethod(GroupKernel,"for finite semigroups", true,
              [IsSemigroup], 0,
    function(M)

    ...

end);
```

Agora um exemplo

```
gap> GroupKernel(popi4);;
gap> time;
570
gap> GroupKernel(popi4);;
gap> time;
0
```

Pelo facto de ser um atributo, o núcleo de um monóide, uma vez calculado, é colocado no "saco" e, durante a sessão GAP a decorrer, não há necessidade de o calcular de novo. É claro que guardar todas estas informações consome memória, pelo que o comando DisableAttributeValueStoring(GroupKernel); pode por vezes ser útil. (Convém no entanto observar que isto não desliga, por exemplo, o guardar automático do atributo Idempotents. Não terá dificuldade em constatá-lo usando o comando KnownAttributesOfObject(objecto);.)

Acima não vimos qual a resposta dada por GroupKernel(popi4). Garanto que nem para os semigrupistas é interessante. Bem mais interessante, para estes, poderá ser saber da distribuição dos elementos do núcleo por $\mathcal{D}$-classes.

```
gap> GroupKernelWithDClassInfo(poi4);
... # lista dos elementos do núcleo
The D-class of Transformation( [ 1 .. 5 ] ) has 1 element which is in K
The D-class of Transformation( [ 1, 2, 4, 5, 5 ] ) has 16 elements
4 of which are in K
The D-class of Transformation( [ 1, 2, 5, 5, 5 ] ) has 36 elements
6 of which are in K
The D-class of Transformation( [ 2, 5, 5, 5, 5 ] ) has 16 elements
4 of which are in K
The D-class of Transformation( [ 5, 5, 5, 5, 5 ] ) has 1 element which is in K
```

O cálculo do núcleo Abeliano de um monóide finito (que tem servido de motivação para muito do meu trabalho) é, usando o algoritmo de que disponho, bastante demorado. Gosto por isso de ir sendo informado daquilo que o computador está a fazer. Criei para o efeito as classes de informação InfoKernel e InfoBasic que incluí, com vários níveis, nos meus programas. Num programa posso, por exemplo, escrever a seguinte linha

```
Info(InfoKernel,2,"I am computing the right Cayley graph of <M>","\n");
```

Posso mudar o *InfoKernel* para o nível 1 escrevendo SetInfoLevel(InfoKernel, 1);. Se escrever info(n) mudo as duas classes que referi para o nível n (por defeito tenho o nível 1), passando então a ser dadas todas as informações de níveis não superiores a n.

```
gap> AbelianKernel(popi4);
#I  I am testing whether the element Transformation( [ 1, 2, 4, 5, 5
 ] ) is in the abelian kernel
... # mais linhas
<semigroup with 13 generators>
```

Mudando agora o nível de informação

```
gap> info(2);
Info Level for InfoKernel and InfoBasic is set to 2
gap> AbelianKernel(poi4);
#I  I am computing the right Cayley graph of <M>

#I  <M> has 16 idempotents

#I   The G-kernel of <M> has 16 elements

#I  I am testing whether the element Transformation( [ 1, 2, 4, 5, 5
 ] ) is in the abelian kernel

#I  The minimal automaton associated to it has size 5

...                        # mais linhas

#I  I am testing whether the element Transformation( [ 1, 3, 5, 5, 5
 ] ) is in the abelian kernel

#I  The minimal automaton associated to it has size 12

#I  Yes, it is in the abelian kernel

#I  Closing for weak conjugation, I found 9 new elements of the
26 elements already found

...                         # mais linhas

<semigroup with 13 generators>
```

A muito útil distribuição por $\mathcal{D}$-classes...

```
gap> AbelianKernelWithDClassInfo(poi4);

... # list of elements in the abelian kernel

The D-class of Transformation( [ 1 .. 5 ] ) has 1 element which is in K
The D-class of Transformation( [ 1, 2, 4, 5, 5 ] ) has 16 elements
4 of which are in K
The D-class of Transformation( [ 1, 2, 5, 5, 5 ] ) has 36 elements
36 of which are in K
The D-class of Transformation( [ 2, 5, 5, 5, 5 ] ) has 16 elements
16 of which are in K
The D-class of Transformation( [ 5, 5, 5, 5, 5 ] ) has 1 element which is in K
```

Posso agora querer saber mais alguma coisa acerca destes núcleos abelianos

```
gap> IsAperiodic(AbelianKernel(poi4));
true
gap> IsAperiodic(AbelianKernel(popi4));
false
```

Pensava-se que os núcleos abelianos dos monóides de uma certa classe à qual popi4 pertence fossem todos aperiódicos. O último exemplo mostra que não. (Ver [DF].)

3.3 Operacões implí citas

As únicas operações implícitas aqui consideradas são as *palavras*, também ditas *operações explícitas*, e as que envolvem potências $\omega + k$, com k inteiro. Devo observar que são quase exclusivamente estas as operações implícitas que aparecem na literatura. (Não entro em detalhes: observo apenas que se x é um elemento de um semigrupo finito S, então x^{ω} é o único idempotente do subsemigrupo gerado por x. O significado de $x^{\omega+k}$ para k positivo é óbvio, sendo não tão óbvio para k negativo: digo apenas que se S for um grupo, então $x^{\omega-1}$ é o inverso de x.) Pode encontrar toda a teoria em [Al].

Uma operação implícita pode ser dada do modo seguinte

```
gap> r:=ImplicitOperationOnnLetters(2,"product",0,
> [ImplicitOperationOnnLetters(2,[],0,[1]),
> ImplicitOperationOnnLetters(2,"omegapower",-5,2)]);
xyˆ(w-5)
gap> ImplicitOperationOnnLetters(2,"power",3,r);
(xyˆ(w-5))ˆ3
```

Podem também, de modo análogo ao já feito com as expressões racionais, ser construídas operações implícitas usando as funções indicadas no exemplo a seguir

```
gap> OmegaPoweIOp(PoweIOp(ProductIOp(r,r),7),-3);
((xyˆ(w-5)xyˆ(w-5))ˆ7)ˆ(w-3)
```

Uma *pseudoidentidade* é uma igualdade formal de operações implícitas. Uma pseudoidentidade é válida num semigrupo finito se para qualquer substituição das variáveis por elementos do semigrupo (um por cada variável) se obtiver uma igualdade. Tal como as variedades (no sentido de Birkhoff) são definidas por identidades, as pseudovariedades (de grande importância no estudo dos semigrupos finitos) são definidas por pseudoidentidades. Nalguns casos, ser capaz de testar pseudoidentidades basta para poder concluir da pertença ou não de um semigrupo a uma pseudovariedade.

Podemos usar o GAP para testar pseudoidentidades

```
gap> w1 := ImplicitOperationOnnLetters(1,"omegapower",1,1);
xˆ(w+1)
gap> w2 := ImplicitOperationOnnLetters(1,"omegapower",0,1);
xˆw
gap> CheckPseudoIdentity(popi4,w1,w2);
false
gap> CheckPseudoIdentity(poi4,w1,w2);
true
```

Este exemplo não testa mais que a aperiocidade ou não dos semigrupos em causa. (A função IsAperiodic faz precisamente isto.)

Exercícios

1. Escreva um programa em GAP que lhe permita encontrar a lista das factorizações (em primos) dos factoriais dos inteiros entre dois inteiros positivos n_1 e n_2 dados.

   Não se esqueça que o programa deve testar, por exemplo, se n_1 e n_2 são inteiros positivos. Crie algumas classes de informação que lhe permitam, se o desejar, ir conhecendo resultados parciais, bem como saber quais os cálculos que o computador está a realizar.

2. Proponha, e resolva, um exercício análogo ao anterior substituindo o cálculo de factoriais, bem como a factorização de inteiros por outras operações.

3. Quais são os objectos com que mais trabalha? Se eles já existem no GAP, também já existem no GAP implementações dos algoritmos que mais usa? Se alguma das respostas é negativa, seria capaz de mudar a situação?

Na página seguinte proponho uma possível solução para o primeiro exercício. Os outros ficam ao cuidado do leitor o qual pode, para o terceiro exercício e dentro das minhas limitações, contar com a minha ajuda.

Uma possível solução do primeiro exercício.

```
DeclareInfoClass("InfoTeste");
DeclareInfoClass("InfoTudo");
#############################################################################
##
#F teste
##
##

teste := function(n1,n2)
    local i, k, L, variavel;

    if not (IsPosInt(n1) and IsPosInt(n2)) or n2 < n1 then
       Error( "<n1> e <n2> devem ser inteiros positivos, com n1 <= n2" );
    fi;

    variavel := 1;
    L := [];
    for i in [n1 .. n2] do
        Info(InfoTeste,1,"Vou calcular o factorial de ", i, "\n");
        k := fac(i);              ##uso a função fac anteriormente escrita
        Info(InfoTeste,2,"Agora vou factoriza-lo", "\n");
        Info(InfoTudo,2,"Ja calculei ", variavel, " factoriais","\n");
        Info(InfoTudo,1, "O factorial de ", i, " é ", "\n");
        PrintFactorsInt(k); Print("\n");

        UniteSet(L,[FactorsInt(k)]);
        variavel := variavel + 1;
    od;
    return L;
end;
```

Bibliografi a

[Al] J. Almeida, "Finite Semigroups and Universal Algebra" World Scientific, Singapore, 1995.

[BL] T. Breuer e S. Linton, *The GAP4 Type System: Organizing Algebraic Algorithms*, in Proceedings of ISSAC 1998, ACM Press

[D:98] M. Delgado, *Abelian pointlikes of a monoid*, Semigroup Forum **56** (1998) 339–361.

[D] M. Delgado, *Commutative images of rational languages and the Abelian kernel of a monoid*, in preparation, 2001.

[DF] M. Delgado and V. H. Fernandes, *Abelian kernels of some monoids of injective partial transformations and an application*, Semigroup Forum **61** (2000) 435–452.

[GAP] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.2* , Aachen, St Andrews, 1999, (`http://www-gap.dcs.st-and.ac.uk/~gap`).

[dG] W. de Graaf, GAP4 *nice and easy*, Universidade de St Andrews, manuscrito.

[H&U] J. E. Hopcroft *e* J. D. Ullman, "Introduction to Automata Theory, Languages and Computation", Addison Wesley, 1979.

[Hu] A. Hulpke, *Using* GAP, *tutorial* dado no ISSAC 2000.