Departamento de Matemática e Engenharias

Emulação da Máquina URM no *Mathematica*

José Laurindo de Góis Nóbrega Sobrinho

Relatório de uma aula prática para satisfação parcial dos requisitos das Provas de Aptidão Pedagógica e Capacidade Científica para habilitação à categoria de Assistente

Emulação da Máquina URM no *Mathematica*

José Laurindo de Góis Nóbrega Sobrinho

(Licenciado em Física pela Universidade da Madeira)

Relatório de uma aula prática para satisfação parcial dos
requisitos das Provas de Aptidão Pedagógica e Capacidade
Científica para habilitação à categoria de Assistente

Universidade da Madeira

Funchal, Outubro 2003

Relatório Realizado sob a Orientação do

Professor Doutor José Manuel Cunha Leal Molarinho Carmo

Professor Catedrático do Departamento de Matemática e Engenharias da Universidade da Madeira

Índice

1 Introdução **5**

2 Apresentação da disciplina **7**

2.1 Enquadramento........ 7

2.2 Programa........ 7

2.3 Plano das aulas teórico-práticas........ 8

3 A Lição **10**

3.1 Enquadramento, objectivos e requisitos........ 10

3.2 Plano da aula........ 11

3.3 Apresentação do problema........ 11

3.4 Sumário da aula........ 13

3.5 Revisão da máquina URM........ 14

3.5.1 Estrutura e funcionamento........ 14

3.5.2 Exemplo: O programa "maior"........ 16

3.6 Alguns preliminares *Mathematica*........ 17

3.6.1 Expressões........ 17

3.6.2 Moldes........ 21

3.6.3 Moldes de sequências de expressões........ 22

3.6.4 Programas URM no *Mathematica*........ 23

3.6.5 Como definir a estrutura da máquina URM no *Mathematica*........ 24

3.6.6 A construção *Switch*........ 28

3.7 A escrita do programa........ 29

3.7.1 Estrutura do programa *executaURM*........ 29

3.7.2 Os argumentos........ 31

3.7.3 Inicialização da máquina URM........ 31

3.7.4 A execução do programa URM........ 34

3.7.5 Devolução do resultado........ 38

3.7.6 O programa *executaURM* completo........ 38

3.8 Utilizando o programa........ 40

3.8.1 Escolha do número de passos a efectuar........ 40

3.8.2 Exemplos........ 41

4 Conclusão **46**

Anexo A **47**

A.1 Codificação de programas URM........ 47

A.2 Descodificação de programas URM........ 50

Bibliografia **53**

1 Introdução

Na cadeira de Paradigmas da Programação os alunos familiarizaram-se com o sistema *Mathematica*. Aprenderam a definir funções por abstracção funcional e por atribuição paramétrica. Praticaram os paradigmas da programação recursiva, imperativa, funcional e por reescrita.

Contrariamente ao que seria desejável acaba sempre por não haver tempo para desenvolver programas mais complexos onde se apliquem os conceitos e as técnicas de programação desenvolvidos ao longo do semestre. Estes programas de aplicação podem no entanto ser introduzidos em fases posteriores do curso no âmbito de outras cadeiras.

É o que acontece, por exemplo, no caso da Teoria da Computabilidade e Complexidade. Em anos anteriores foram facultados aos alunos desta cadeira alguns programas em linguagem *Mathematica* capazes de resolver questões específicas referentes à matéria leccionada. Esses programas consistiam no emulador da máquina URM, no codificador de programas URM e no descodificador de programas URM.

Ao facultar o emulador da máquina URM pretendia-se dar aos alunos uma ferramenta adicional que poderiam utilizar, extra-aula, para complementar o seu estudo testando os programas URM apresentados nas aulas bem como outros que viessem a construir. Ao facultar o codificador e o descodificador de programas URM pretendia-se que os alunos constatassem a diferença entre o apenas se pretender garantir que é possível codificar e descodificar programas URM e os aspectos ligados à procura de codificações eficientes desses programas (pois os alunos facilmente verificarão que a codificação dada nas aulas teóricas não seria prática nem tal era uma preocupação no contexto desta cadeira).

Seria no entanto proveitoso para os alunos se fossem eles próprios a construir os referidos programas. Assim eles iriam, para além de praticarem a programação em *Mathematica*, interiorizar de forma mais marcante a matéria da disciplina de Teoria da Computabilidade e Complexidade.

Os programas descritos anteriormente poderiam ser construídos em duas aulas práticas que teriam naturalmente lugar numa sala de informática. A construção do emulador URM corresponderia à segunda aula teórico-prática. Os restantes programas seriam construídos na última aula teórico-prática.

Este relatório descreve a primeira das duas aulas (emulador da máquina URM) estando dividido em quatro partes, sendo a primeira a presente introdução.

Na segunda parte é feita uma breve apresentação da disciplina de Teoria da Computabilidade e Complexidade.

Na terceira parte é apresentada a lição, focando inicialmente o seu enquadramento na disciplina, os seus objectivos e requisitos.

Finalmente, na quarta parte, apresentam-se as conclusões.

Os programas *Mathematica* a construir na segunda aula prática são apresentados em anexo.

2 Apresentação da disciplina

2.1 Enquadramento

A disciplina de Teoria da Computabilidade e Complexidade é uma cadeira obrigatória do 2º ano das licenciaturas em Engenharia Informática e Ensino de Informática da Universidade da Madeira[1]. Em termos de plano de estudos enquadra-se na área da Teoria da Computação.

A carga horária semanal é de 3 horas teóricas (divididas por duas aulas de 1,5 horas) e uma aula teórico-prática de duas horas.

A cadeira é leccionada desde o ano lectivo 1999/2000. Nos três primeiros anos o responsável pela cadeira foi o Professor Doutor José Carmo, Professor Catedrático do Departamento de Matemática e Engenharias da Universidade da Madeira. No ano lectivo 2002/2003 a responsabilidade da cadeira passou para o Professor Doutor Eduardo Fermé, Professor Auxiliar do Departamento de Matemática e Engenharias da Universidade da Madeira.

A minha ligação com a cadeira de Teoria da Computabilidade e Complexidade remonta aos anos lectivos de 2000/2001 e 2001/2002 durante os quais leccionei a componente teórico-prática aos alunos da licenciatura em Ensino de Informática[2]. Quanto à cadeira de Paradigmas da Programação leccionei a componente prática nos anos lectivos de 1999/2000, 2001/2002 e 2002/2003.

2.2 Programa

O programa da cadeira foi planeado para um semestre de 14 semanas efectivas de aulas, correspondendo a 28 aulas teóricas e a 13 aulas teórico-práticas.

Os traços gerais do programa da cadeira são os apresentados a seguir:

---

[1] A partir do ano lectivo 2003-2004 passa também a ser obrigatória na licenciatura em Matemática.
[2] A licenciatura em Engenharia Informática só se iniciou em 2001-2002.

- Introdução aos conceitos fundamentais da computação, como: sistemas formais, máquinas abstractas, computabilidade, decidibilidade e complexidade.

- Funções computáveis: máquina de registos URM e funções URM-computáveis.

- Outras abordagens à computabilidade: funções recursivas e primitivas recursivas, máquina de Turing e funções Turing-computáveis.

- Tese de Church.

- Enumeração de programas e funções.

- Programas universais.

- Decidibilidade, total e parcial, e indecidibilidade.

- Conjuntos recursivos e recursivamente enumeráveis.

- Introdução à problemática da complexidade: P e NP.

2.3 Plano das aulas teórico-práticas

Para o ano lectivo 2003/2004 estão previstas 13 aulas teórico-práticas (26 horas) onde já se incluem as duas aulas práticas referidas neste relatório. O plano é o indicado a seguir:

1 Construção de programas URM e respectivos fluxogramas
2 Construção do emulador da máquina URM (no *Mathematica*)
3 Mais exemplos da construção de programas URM.
Demonstração de que as instruções URM de transferência, embora muito

úteis, não são indispensáveis.

4 Demonstração de que certos predicados são decidíveis.
Construção de programas URM com "subprogramas".

5 Prática de definição de funções a partir das funções básicas por composição de funções e recursão: exemplos vários.

6 Demonstração da computabilidade de certas funções, e da decidibilidade de certos predicados, por aplicação das técnicas dadas na última aula teórica (soma limitada, produto limitado e minimização limitada).

7 Máquina de Turing: descrição do seu funcionamento, conceitos e convenções usuais.

8 Programação na Máquina de Turing: exemplos.

9 Programação na Máquina de Turing: exemplos.

10 Demonstração da indecibilidade de certos problemas, por aplicação dos métodos da diagonal e da redução.

11 Demonstração da indecibilidade de certos problemas, por aplicação do teorema s-m-n e do teorema de Rice.

12 Predicados parcialmente decidíveis: exercícios.
Conjuntos recursivos e enumeráveis: exercícios.

13 Construção do codificador e descodificador de programas URM (no *Mathematica*)

3 A Lição

3.1 Enquadramento, objectivos e requisitos

O objectivo de fundo desta aula é a construção de um simulador da máquina URM em linguagem *Mathematica*. Não se pretende apenas facultar aos alunos um programa que lhes permita testar o funcionamento de programas URM mas também fazer com que sejam eles próprios a construírem esse programa. Pretende-se com isso que os alunos interiorizem bem a estrutura, funcionamento e programação da máquina URM.

Esta lição tem também como objectivo apresentar aos alunos mais um exemplo de aplicação da linguagem *Mathematica*, complementar àqueles que foram mostrados nas aulas de Paradigmas da Programação.

Como objectivo adicional pretende-se o aprofundar, por parte dos alunos, da linguagem *Mathematica*, mostrando-lhes, com mais este exemplo concreto, que o sistema *Mathematica* é uma excelente ferramenta de trabalho.

Ao frequentarem esta lição suplementar os alunos já devem conhecer a estrutura da máquina URM, o tipo de instruções suportadas por esta e já devem ter escrito alguns programas URM nas aulas Teórico-Práticas.

Os alunos devem também dominar algumas técnicas de programação, no âmbito da linguagem *Mathematica*, resultantes da frequência das aulas de Paradigmas da Programação no ano anterior. Em particular os alunos estão familiarizados, por exemplo, com a estrutura de composição iterativa *While* e com a estrutura de composição alternativa *If*.

Esta lição pode também ser adaptada à disciplina de Paradigmas da Programação. Figuraria nesse caso como uma das últimas aulas onde se apresentam exemplos de aplicação das técnicas de programação desenvolvidas ao longo do semestre. Este seria, provavelmente, o primeiro contacto dos alunos com a máquina URM pelo que teriam de ser considerados apenas exemplos de programação URM bastante simples.

3.2 Plano da Aula

As aulas teórico-práticas e práticas têm, regra geral, uma duração de duas horas. No plano seguinte indica-se, para cada uma das partes em que está dividida a aula, o intervalo de tempo correspondente (em minutos).

1. Enunciar o problema, apresentar e descrever o plano (sumário) da aula......**(10m)**
2. Revisão ou introdução de algumas das funções *Mathematica* a utilizar.......**(30m)**
3. Elaboração do programa *executaURM*...**(45m)**
4. Teste do programa com alguns exemplos...**(35m)**

Os intervalos indicados são aqueles que se julgam ser os mais adequados tendo em conta o tempo disponível e o conteúdo da aula. Podem no entanto ser alterados com o decorrer da aula. Pode, por exemplo, economizar-se algum tempo nos pontos 2 e 3 em benefício do ponto 4 onde se poderão testar mais exemplos.

3.3 Apresentação do Problema

É apresentado aos alunos o enunciado do problema que pretendemos resolver nesta aula:

Problema:

> Definir uma função *Mathematica*, de nome *executaURM*, que recebendo um programa URM, p, e uma sequência de n números naturais $x_1,...,x_n$, retorna o resultado de aplicar a $x_1,...,x_n$ a função n-ária calculada pelo programa p.
>
> Para evitar a possibilidade de não terminação do cálculo deverá ainda ser indicado, juntamente com os argumentos anteriores, um inteiro positivo i. Assim se não for obtido um resultado num número de passos

não superior a *i*, o programa deverá parar retornando a mensagem: *" O programa não parou ao fim de* " *i* "*passos.*".

Ilustração da evocação da função *executaURM*:

Por forma a evitar quaisquer dúvidas, relativamente ao pretendido, é conveniente ilustrar a evocação da função *executaURM* com dois exemplos como os que se seguem:

Exemplo 1:

executaURM [maior,7,8,100]

Neste caso estamos a supor que "maior" é uma variável *Mathematica* onde está guardado um programa URM capaz de calcular a função:

$$f(x,y)=\begin{cases} x, & x \geq y \\ y, & x < y \end{cases}$$

Os números 7 e 8 são os argumentos da função e o número 100 é o número de passos findos os quais o programa deve parar (se ainda não o tiver feito antes). A forma de codificar programas URM no *Mathematica* será discutida mais adiante.

Se o programa terminar antes de se ultrapassarem os 100 passos então deverá ser devolvida a resposta desejada:

8

Note-se que existe uma infinidade de programas URM capazes de calcular a função anterior. O programa que consideraremos mais adiante terminaria, para os argumentos deste exemplo, muito antes de serem atingidos os 100 passos. Pode no entanto imaginar-se programas URM que também calculem a mesma função *f* mas que, por serem mais extensos, requeiram um número de passos superior para que a execução termine.

<u>Exemplo 2</u>:

executaURM[CicloInfinito,6,7,9,100]

Neste exemplo "CicloInfinito" é uma variável *Mathematica* onde está guardado, por exemplo, o programa URM cuja única instrução é J[1,1,1]. Neste caso, independentemente do número e valor dos argumentos considerados o resultado será a mensagem:

O programa não terminou ao fim de 100 passos.

3.4 Sumário da aula

Depois dos alunos terem tomado conhecimento do problema a resolver durante esta aula pode apresentar-se, em jeito de sumário, o plano proposto para o decurso da mesma, seguindo-se depois uma breve explicação de cada um dos pontos focados.

<u>Sumário</u>

Revisão ou introdução de algumas das funções Mathematica a utilizar.
Elaboração do programa ***executaURM.***
Teste do programa com alguns exemplos.

O programa *executaURM* será escrito no *Mathematica*. Os alunos já tomaram conhecimento com o sistema *Mathematica* nas aulas de Paradigmas da Programação durante a frequência do 1ºsemestre do 1ºano da sua licenciatura. Durante essas aulas os alunos aplicaram os paradigmas da programação recursiva, imperativa, funcional e por reescrita. Aprenderam, entre outros assuntos, a definir funções por abstracção funcional e por atribuição paramétrica diferida.

Vamos rever aqui alguns desses conceitos. Serão revistas algumas das funções *Mathematica* utilizadas e introduzidas outras novas que se afiguram úteis para a

resolução do problema em questão. Discutiremos a forma de representar as instruções e os programas URM, bem como a forma de representar a estrutura da máquina URM.

Esta aula pode ser vista, por parte dos alunos, como uma aula de aplicação do sistema *Mathematica* onde será desenvolvido um programa com uma complexidade talvez um pouco superior à generalidade dos elaborados no decurso das aulas de Paradigmas da Programação, mas perfeitamente factível com os conhecimentos que os alunos adquiriram nessa disciplina.

Com os esclarecimentos anteriores os alunos devem ser capazes de construir o programa *executaURM*.

Finalmente pretende-se testar o programa *executaURM* com alguns programas URM. Alguns desses programas já foram construídos na 1ª aula teórico-prática, outros serão ser desenvolvidos nesta aula e outros ainda propostos para trabalho de casa.

3.5 Revisão da máquina URM

3.5.1 Estrutura e funcionamento

Os alunos chegam a esta aula depois de já terem estudado a máquina URM nas aulas teóricas e depois de terem tido a primeira aula teórico-prática dedicada à escrita de programas URM. Nesta secção do relatório é indicado o que se está a assumir que os alunos já conhecem sobre a máquina URM.

URM - **U**nlimited **R**egister **M**achine

- Existe um número infinito de registos designados por R_1, R_2, R_3, ...

- Cada registo contém um número natural. O número contido no registo R_n é normalmente designado por r_n.

	R_1	**R_2**	**R_3**	**R_4**	**R_5**	**...**
URM	**r_1**	**r_2**	**r_3**	**r_4**	**r_5**	**....**

- O conteúdo de cada registo pode ser alterado pela máquina URM em resposta a determinadas instruções. As Instruções reconhecidas pela máquina URM são as indicadas na tabela seguinte (onde *n*, *m* e *q* são números naturais positivos).

Nome da Instrução	**Instrução**	**Resultado**
Zero	Z(n)	$r_n \rightarrow 0$
Sucessor	S(n)	$r_n \rightarrow r_n+1$
Transferência	T(m,n)	$r_n \rightarrow r_m$
Salto	J(m,n,q)	Se $r_m=r_n$ então o programa continua na instrução nº q, caso contrário continua na instrução seguinte.

- Um programa URM é constituído por uma sequência finita de instruções: $I_1, I_2, I_3,...,I_s$.

- A execução do programa começa sempre pela instrução I_1.

- Se a instrução I_k não for uma instrução de salto então será executada a seguir a instrução I_{k+1}. Se I_k for da forma $J(m,n,q)$ então a instrução a executar a seguir será I_q se $r_m=r_n$ ou I_{k+1} no caso contrário.

- A execução de um programa, $I_1, I_2, I_3,...,I_s$, termina quando o índice da instrução a executar a seguir for superior a *s* (ou seja quando a instrução a executar a seguir não existir). Se isso nunca acontecer então o programa nunca termina.

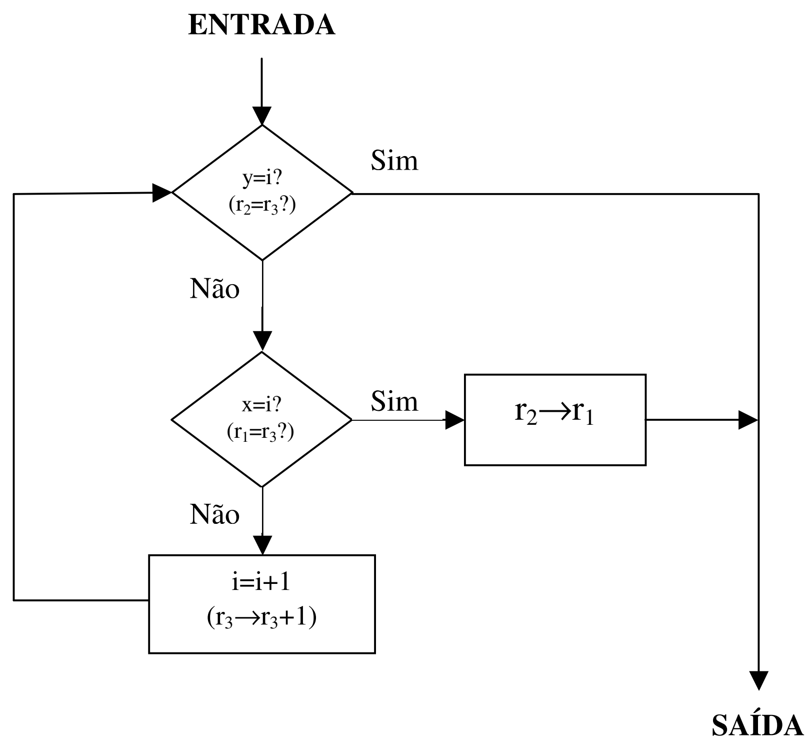
- Para que a máquina URM efectue uma computação é necessário fornecer, para além de um programa, uma configuração inicial. Esta configuração consiste numa sequência de números naturais $a_1, a_2, a_3,...$ nos registos $R_1, R_2, R_3, ...$

- Quando a execução de um programa termina o conteúdo dos registos determinam a configuração final da máquina.

- Se utilizarmos a máquina URM para calcular funções de $\mathbb{N}^n$ em $\mathbb{N}$ então os argumentos estão inicialmente nos registos $R_1,...,R_n$ guardando os restantes registos o valor zero. Por convenção o resultado deverá ser devolvido no registo R_1.

3.5.2 Exemplo: O Programa "maior"

Vamos construir um programa URM capaz de calcular a função *f* introduzida na ilustração feita no enunciado do problema (Secção3.3). A este programa URM, que utilizaremos como exemplo ao longo da aula, daremos o nome de "*maior*".



	R_1	R_2	R_3	R_4	R_5	...
URM	**x**	**y**	**i**	**0**	**0**	**....**

I1 J(2,3,6)
I2 J(1,3,5)
I3 S(3)
I4 J(1,1,1)
I5 T(2,1)

3.6 Alguns preliminares *Mathematica*

Nesta secção são revistos os conhecimentos que os alunos já têm acerca de *expressões* e *moldes* sendo depois introduzida a noção de *moldes de sequências*. É discutida a codificação de programas URM bem como a implementação da estrutura da máquina URM no *Mathematica*. Para isso é também introduzida a estrutura de composição alternativa *Switch*

3.6.1 Expressões

Basicamente tudo o que se escreve em *Mathematica* é uma expressão. Por exemplo, fórmulas matemáticas, gráficos, listas e polinómios, embora sejam conceitos distintos, são representados pelo *Mathematica* de uma forma uniformizada como expressões.

Por exemplo, *x+y*, é uma expressão. Para saber a forma como o *Mathematica* armazena internamente uma expressão podemos recorrer à função *FullForm*:

```
? FullForm
```

```
FullForm[expr] prints as the full form of expr, with no special syntax.
```

```
FullForm[x+y]
```

```
Plus[x,y]
```

Podemos também recorrer à função *Head* para determinar o tipo de entidade que está à cabeça de uma expressão:

```
?Head
```

Head[expr] gives the head of expr.

```
Head[x+y]
```

Plus

A expressão *x+y* é composta por três partes: um operador (*Plus*) e dois argumentos (*x* e *y*). Recorde-se que em *Mathematica* os argumentos de funções são colocados entre parênteses rectos. Para aceder às diferentes partes de uma expressão podemos utilizar o selector de componente *[[]]*. Assim:

```
(x+y)[[1]]
```

x

```
(x+y)[[2]]
```

y

```
(x+y)[[0]]
```

Plus

Ou então recorrendo à função *Part*:

```
?Part
```

expr[[i]] or Part[expr, i] gives the ith part of expr. expr[[-i]] counts from the end. expr[[0]] gives the head of expr. expr[[i, j, ...]] or Part[expr, i, j, ...] is equivalent to expr[[i]] [[j]] ... . expr[[{i1, i2, ... }]] gives a list of the parts i1, i2, ... of expr.

```
Part[x+y,1]
```

x

```
Part[x+y,2]
```

```
y
```

```
Part[x+y,0]
```

```
Plus
```

Podemos optar por representar no *Mathematica* as instruções URM por meio de expressões do tipo:

Z[n]
S[n]
T[m,n]
J[m,n,q]

onde *m*, *n* e *q* são números naturais positivos, facilitando assim o acesso às suas componentes, tirando partido das operações que o *Mathematica* disponibiliza para o manuseamento das expressões. Note-se que a única diferença em relação à escrita usual (seguida nas aulas teóricas e teórico-práticas) é a substituição dos parênteses curvos por parênteses rectos.

Consideremos, por exemplo, a instrução *Z(5)*, que representaremos no *Mathematica* pela expressão *Z[5]*. Seja então:

```
inst = Z[5];
```

Para identificar o tipo de instrução presente em *inst* fazemos:

```
Head[inst]
```

```
Z
```

ou ainda:

```
inst[[0]]
```

```
Z
```

ou:

```
Part[inst,0]
```

```
Z
```

Uma vez identificado o tipo de instrução sabemos o número de argumentos que devem estar presentes. No caso da instrução Zero deve existir apenas um argumento (o índice do registo cujo conteúdo deve ser igualado a zero). Continuando com o exemplo anterior teríamos:

```
inst[[1]]
```

```
5
```

ou ainda:

```
Part[inst,1]
```

```
5
```

Consideremos, como segundo exemplo, a instrução de salto *J(2,5,10)*. Neste caso temos uma expressão composta por três partes:

```
inst=J[2,5,10];
```

```
Head[inst]
```

```
J
```

```
inst [[1]]
```

```
2
```

```
inst[[2]]
```

```
5
```

```
inst[[3]]
```

```
10
```

3.6.2 Moldes

A especificação dos argumentos admissíveis, para uma determinada expressão, é obtida através da noção de molde. O molde universal, representado por _, é verificado por qualquer expressão. Consideremos, por exemplo, a função *f* definida como se segue:

```
f[_]:={1,2,3};
```

Se esta função receber como argumento uma expressão qualquer então devolverá como resultado a lista {1,2,3}, caso contrário, a função fica por avaliar[3]. Ilustremos com dois exemplos:

```
f[2]
```

```
{1,2,3}
```

Também podemos especificar qual deve ser a cabeça do argumento e etiquetar o molde (dar um nome ao argumento para o poder utilizar). Consideremos o exemplo seguinte:

```
g[x_Integer]:={x,x+1,x+2};
```

A função *g* só será executada caso a cabeça do respectivo argumento seja um inteiro. Se assim for, esse inteiro é etiquetado com a letra *x*, sendo o resultado da avaliação a lista {x,x+1, x+2} como se ilustra a seguir:

```
g[6]
```

```
{6,7,8}
```

```
g[6.7]
```

```
g[6.7]
```

[3] A função fica por avaliar porque não existe nenhuma regra de reescrita que a permita reescrever. Por exemplo f[2,6] não é avaliado.

3.6.3 Moldes de sequências de expressões

No *Mathematica* uma sequência traduz-se por um qualquer número, eventualmente nulo, de expressões, separados por vírgulas se forem em número superior a um. Existem também moldes próprios para representar sequências de expressões:

- __ - molde satisfeito por uma sequência de uma ou mais expressões;
- __h - molde satisfeito por uma sequência de uma ou mais expressões, cada uma delas com a cabeça h;
- ___- molde satisfeito por uma sequência de zero ou mais expressões;
- ___h - molde satisfeito por uma sequência de zero ou mais expressões, cada uma delas com a cabeça h;

Consideremos a seguinte entrada:

```
f1[___]:=5;
f2[___Integer]:=5;
g1[x___Integer]:=Length[{x}];
g2[x__Integer]:=Length[{x}];
```

A função *f1* recebe uma sequência zero ou mais expressões de qualquer tipo devolvendo como resultado o número 5. A função *f2* difere de *f1* por requerer que a sequência seja constituída apenas por inteiros (se assim não for a função fica por avaliar). Verifiquemos isto com os exemplos seguintes:

```
f1[]
```

```
5
```

```
f1[7.2]
```

```
5
```

```
f2[]
```

```
5
```

f2[7.2]

f2[7.2]

f2[1,2]

5

f2[6]

5

f2[2.1,5]

f2[2.1,5]

A função *g1* ao receber uma sequência de zero ou mais inteiros devolve o número de elementos existentes nessa sequência (através da transformação dessa sequência numa lista formada pelos elementos da sequência). Para que se possa fazer referência à sequência, recebida como argumento, é necessário etiquetar a mesma. Neste caso utilizamos como etiqueta a letra *x*. A função *g2* é análoga a *g1* diferindo apenas no facto de ser avaliada para sequências de um ou mais inteiros. Considere-se como exemplo o seguinte:

g1[5,6,7,8,9,10]

6

g1[]

0

g2[]

g2[]

3.6.4 Programas URM no *Mathematica*

O *Mathematica* disponibiliza a estrutura *List* onde se podem guardar colecções finitas de elementos não necessariamente todos do mesmo tipo.

?List

{e1, e2, ... } is a list of elements.

Um programa URM consiste numa sequência de instruções $I_1, I_2, I_3, ..., I_s$. Como vimos anteriormente as instruções URM podem representar-se por expressões *Mathematica*. Podemos assim guardar a sequência de instruções que compõem um programa URM na forma de lista. No caso do programa "*maior*" poderíamos definir a lista:

maior = {J[2,3,6], J[1,3,5], S[3], J[1,1,1], T[2,1]};

Note-se que o número de instruções do programa é exactamente igual ao número de elementos da lista (dado por *Length*[maior]).

3.6.5 Como definir a estrutura da Máquina URM no *Mathematica*

Como poderemos guardar a estrutura de uma máquina URM no sistema *Mathematica*? Relembre-se que a estrutura de uma máquina URM consiste num conjunto infinito de registos cada qual guardando um número natural.

Uma possível solução apontaria para a construção de uma lista de números naturais cada qual correspondendo a uma posição de memória. Acontece que uma lista *Mathematica* consiste numa sequência finita de elementos e a máquina URM é composta por um número infinito de registos. Assim as listas não são a solução mais adequada.

Poderíamos também optar, ainda dentro do âmbito das listas, por uma representação compactada dos registos da máquina URM. Essa representação consistiria numa lista de listas em que cada sublista corresponderia a um registo cujo valor fosse diferente de zero. Assim cada sublista guardaria dois valores naturais, sendo o primeiro deles o índice do registo (por exemplo) e o segundo o respectivo valor. Utilizando as várias funções disponibilizadas pelo *Mathematica* poderíamos aceder e alterar a informação contida nessa lista.

Existe no entanto uma terceira hipótese muito mais flexível. Podemos, tirando partido dos mecanismos de reescrita suportados pelo *Mathematica*, definir uma família de variáveis ("variável indexada"), de nome *m*, tal que *m[i]* devolva o valor do registo R_i num dado instante.

Por exemplo quando se pretende executar o programa maior com os argumentos (7,8) o estado inicial da máquina deverá ser:

7 8 0 0 0 0 0 0 0 ....

podendo ser descrito pela família de variáveis:

$$m(n) = \begin{cases} 7 & , n = 1 \\ 8 & , n = 2 \\ 0 & , n > 2 \end{cases}$$

Esta família de variáveis pode definir-se no *Mathematica*, de forma muito simples, por atribuição imediata. Continuando com o exemplo anterior teríamos:

```
m[1]=7;

m[2]=8;

m[_]=0;
```

Cada uma destas instruções é designada por regra. Na última regra estamos a utilizar o chamado *molde universal* _. A regra *m[_]=0* determina que o valor da função será zero para qualquer argumento excepto para aqueles em relação aos quais tenha sido definida alguma regra mais específica (como acontece neste caso para o 1 e para o 2).

As três regras anteriores permitem então estabelecer a configuração inicial da maquina URM para a computação de f(7,8). Procuremos confirmar este facto mandando avaliar o conteúdo de alguns registos:

```
m[1]
```

```
7
```

```
m[2]
```

8

```
m[3]
```

0

```
m[123456789]
```

0

Durante a execução do programa URM a configuração inicial será naturalmente alterada. As alterações que podem ocorrer sobre o conteúdo de um dado registo são as seguintes:

- igualar o valor do registo a zero (resulta da execução da instrução Zero). Por exemplo a instrução *Z[i]* será simulada aplicando a regra *m[i]=0*;

- incrementar o valor do registo (resulta da aplicação da instrução Sucessor). Por exemplo *S[i]* será simulada aplicando a regra *m[i]=m[i]+1*;

- substituir o valor de um registo pelo valor de outro (resulta da aplicação da instrução de Transferência). Por exemplo *T[i,j]* será simulada por *m[j]=m[i]*;

Procuremos testar, no *Mathematica*, cada uma das operações descritas anteriormente. Antes porém verifiquemos a configuração actual da máquina:

```
?m
```

```
Global`m

m[1] = 7

m[2] = 8

m[_] = 0
```

Incrementemos, por exemplo, o valor contido no registo R_5 (equivalente a executar *S[5]*):

```
m[5]=m[5]+1;
```

Verifiquemos de novo a configuração da máquina:

```
?m
```

```
Global`m

m[1] = 7

m[2] = 8

m[5] = 1

m[_] = 0
```

Transfira-se o conteúdo do registo R_2 para o registo R_3 (equivalente a *T[2,3]*) e verifique-se novamente a configuração da máquina:

```
m[3]=m[2];
```

```
?m
```

```
Global`m

m[1] = 7

m[2] = 8

m[3] = 8

m[5] = 1

m[_] = 0
```

Igualemos a zero o conteúdo de R_2 (equivalente a *Z[2]*) e voltemos a ver a configuração da máquina:

```
m[2]=0;
```

```
?m
```

```
Global` m

m[1] = 7

m[2] = 0

m[3]=8

m[5]=1

m[_] = 0
```

Com os testes descritos (juntamente com outros que os alunos possam realizar) verifica-se que a função *m* descreve bem a estrutura da máquina URM.

3.6.6 A construção *Switch*

Uma vez conhecida a cabeça da instrução URM a executar de seguida há que agir em conformidade. Estamos aqui perante um problema de *composição alternativa*. A única estrutura de composição alternativa que os alunos aplicaram durante a cadeira de Paradigmas da Programação foi o *If*.

```
?If
```

If[condition, t,f] gives t if condition evaluates to True, and f if it evaluates to False. If[condition, t, f,u] gives u if condition evaluates to neither True nor False.

Podemos optar por utilizar uma estrutura deste tipo. Como existem quatro tipos de instruções teríamos de utilizar um sistema de três *If's* encadeados como se procura ilustrar a seguir:

```
If[Head[x] == Z, < executa Zero >,
  If[Head[x] == S , < executa Sucessor >,
    If[Head[x] == T, < executa Transferência >, < executa Salto >]
    ]
  ]
```

Note-se que todas as condições têm a forma de uma comparação entre uma mesma expressão (*Head[x]*) e as constantes Z, S e T. Como estas situações são muito frequentes em programação, e a escrita e leitura de *If's* encadeados pode tornar-se numa tarefa aborrecida e por vezes confusa, o *Mathematica* disponibiliza a estrutura *Switch*.

```
?Switch
```

Switch[expr, form1, value1, form2, value2, ...] evaluates expr, then compares it with each of the formi in turn, evaluating and returning the valuei corresponding to the first match found.

Note-se ainda que *value1*, *value2*,... podem ser instruções simples ou sequências de instruções (composição sequencial). Adaptando a estrutura *Switch* ao nosso problema, vamos substituir a cadeia de *If's* anteriores por:

```
Switch[Head[x],

  Z, < executa Zero >,
  S, < executa Sucessor >,
  T, < executa Transferência >,
  J, < executa Salto >

  ]
```

3.7 A escrita do programa

3.7.1 Estrutura do programa *executaURM*

Depois da exposição e discussão anterior podemos partir para a escrita do programa *executaURM*. Tratando-se de um programa com uma complexidade ligeiramente superior à maioria dos apresentados ao longo das aulas de Paradigmas da Programação convém antes de mais começar por subdividir o mesmo. Pode então apresentar-se o seguinte esquema:

executaURM[< Argumentos >] :=
(<Inicialização da máquina URM >;
< Execução do programa URM >;
< Devolução do resultado >;)

A escrita do programa *executaURM* deve ser levada a cabo pelos alunos. Estes podem trabalhar individualmente ou em grupos de dois (tudo dependerá do número de alunos presentes e do número de terminais existentes na sala).

Nesta fase o docente deve deslocar-se pelos vários grupos esclarecendo possíveis dúvidas e observando os respectivos desempenhos. Deve dar-se liberdade aos alunos para que possam desenvolver as suas ideias, sem contudo deixar de os orientar caso se verifique que o caminho que pretendam seguir não é o mais correcto.

É natural que nem todos os alunos ou grupos tenham o mesmo desempenho. Caso o docente verifique que algum ou alguns dos grupos estão a atrasar-se demasiado deve ajudá-los para que tenham uma melhor progressão.

Nas páginas seguintes são apresentadas e discutidas as várias partes do programa *executaURM*. Numa situação ideal, em que todos os alunos consigam por si mesmos concretizar a escrita do programa, o docente poderá omitir, em parte ou na totalidade, a discussão relativa à escrita do programa (Secções 3.7.2 a 3.7.5), tecendo apenas algumas considerações que ache oportunas no momento.

Numa situação, provavelmente mais comum, em que alguns alunos apresentem algumas dificuldades, o docente deve ir expondo e discutindo as diferentes partes do programa à medida que os alunos vão progredindo. Toda esta acção deve ser feita de forma a gerir do melhor modo os 45 minutos previstos para esta parte da aula. Embora o objectivo de fundo da aula seja a construção do programa *executaURM* é extremamente importante que no final reste algum tempo para que se possa testar o funcionamento do mesmo.

3.7.2 Os argumentos

Uma hipótese para a emulação da máquina URM era o programa *Mathematica* receber através de três argumentos o programa URM, a configuração inicial (recebendo o nome da família de variáveis que guarda o estado da máquina) e o número de passos. Nessa opção a configuração inicial era construída à parte. Na opção aqui considerada é o próprio programa *Mathematica* que constrói a configuração inicial. Não se pretende apenas emular a execução de um programa URM, mas sim emular a função que aquele calcula, de qualquer aridade.

O programa *executaURM*, a construir nesta aula, deve então receber como argumentos:

- uma lista contendo um programa URM.
- um ou mais números naturais (argumentos do programa URM).
- um inteiro positivo (número máximo de passos a efectuar na execução).

Podemos então escrever:

executaURM[p_, args__, i_] := ...

onde estamos a fazer uso dos moldes **_** e **__** (Secções 3.6.2 e 3.6.3). Não nos vamos preocupar aqui em verificar se os argumentos são ou não do tipo correcto.

3.7.3 Inicialização da máquina URM

A inicialização da Máquina URM faz-se igualando os valores dos registos R_1 a R_n aos n valores contidos na sequência designada por *args* e igualando os restantes registos a zero. Por exemplo, se foi feita a evocação:

```
executaURM[prog, 6, 5, 8, 10, 16]
```

devemos estabelecer a seguinte configuração inicial:

R_1	R_2	R_3	R_4	R_5	R_6	
6	5	8	10	0	0	...

Para isso a função *m*, descrita anteriormente, deve ser definida através da seguintes regras de atribuição imediata:

```
m[1]=6;
m[2]=5;
m[3]=8;
m[4]=10;
m[_]=0;
```

Os argumentos do programa URM são recebidos, pelo programa *Mathematica executaURM*, através de uma sequência, de um ou mais números naturais, que denotamos por *args*. Convém transformar esta sequência numa lista[4]. Para isso podemos considerar:

argsList={args}

ou equivalentemente:

argsList=List[args]

De forma a copiar os *n* valores contidos em *argsList* para os *n* primeiros registos da máquina podemos escrever um ciclo *While*. Esta estrutura de composição iterativa *While* foi utilizada por diversas vezes, nas aulas práticas de Paradigmas da Programação, aquando do estudo da programação imperativa, sendo por isso perfeitamente conhecida dos alunos. Relembremos, no entanto, a estrutura deste tipo de ciclos:

[4] Este deve ser dos únicos aspectos em que os alunos devem precisar verdadeiramente de ajuda.

```
< Inicialização >
While [ < Guarda >,
        < Acção >;
        < Progresso >
        ]
```

O número de iterações do ciclo é dado pelo comprimento da lista *argsList* pelo que podemos considerar a variável auxiliar:

n=Length[argsList]

Para controlar o progresso do ciclo utilizaremos uma variável, *r*, que irá de *1* (índice do primeiro elemento da lista) até *n* (índice do último elemento da lista). Podemos então considerar como inicialização do ciclo:

r=1;

A guarda do ciclo será naturalmente:

r≤n

e o progresso:

r=r+1

Resta considerar a acção. Em cada iteração do ciclo pretende-se atribuir a *m[r]* o r-ésimo elemento da lista *argsList*. Podemos então escrever a acção na forma:

m[r]=argsLists[[r]]

Para terminar a configuração inicial da máquina URM resta declarar que todos os restantes registos, isto é, todos os registos R_m com $m>n$ são iguais a zero. Para isso utilizaremos, como já foi exposto, a regra:

m[_]=0;

Finalmente podemos escrever o bloco de código correspondente à concretização da configuração inicial da máquina URM:

```
argsList = List[args];
n = Length[argsList];

r = 1;

While[r <= n,

    m[r] = argsList[[r]];

    r = r + 1

];

m[_] = 0;
```

3.7.4 A execução do programa URM

Consideremos o bloco onde ocorre a execução do programa URM. A execução pode ser controlada por um ciclo *While*. Cada iteração do ciclo corresponde à execução de um passo do programa URM.

O número máximo de iterações permitido a esse ciclo *While* é dado pelo parâmetro *i* (número máximo de passos a considerar na execução do programa URM). Se o programa URM terminar num número de passos *r*, com $r<i$, o ciclo deverá terminar findas *r* iterações. Precisamos assim de duas variáveis:

k **-** número ou índice da instrução a executar de seguida. Se essa instrução não existir então o ciclo deverá terminar (uma vez que terminou também o programa

URM). A variável *k* deve ser inicializada com o valor 1 pois é sempre esse o número da primeira instrução a executar em qualquer programa URM.

r **-** número de passos (iterações) executados até um dado instante. Se *r* exceder o valor do parâmetro *i* então o ciclo deve terminar. Esta variável poderá ser inicializada com o valor zero indicando assim que ainda não foi executado qualquer passo.

A *inicialização* poderá ser escrita como se segue:

k=1;
r=0;
s=Length[p];

Introduzimos aqui também uma variável auxiliar s com o objectivo de guardar o número de instruções, ou seja, o comprimento do programa URM. O programa URM é recebido na forma de lista *Mathematica* (como descrito anteriormente) através do parâmetro *p*. O número de instruções do programa URM é então dado pelo comprimento (*Length*) da lista *p*.

A *guarda* do ciclo pode ser escrita, na forma de uma conjunção, como se segue:

```
r ≤ i && k ≤ s
```

O passo do ciclo será executado apenas quando a guarda, avaliada no início de cada iteração, for verdadeira. A guarda será verdadeira se ainda não tiver sido ultrapassado o número máximo de passos permitido ($r \leq i$) e se existir uma instrução para executar de seguida ($k \leq s$). Se uma destas condições falhar então o passo do ciclo já não será executado.

O progresso assegura a terminação de uma composição iterativa. Neste caso existem duas variáveis do progresso (*k* e *r*). A actualização da variável *k* depende do tipo de instrução URM em execução. Por sua vez a actualização da variável *r*, que deve ser feita em cada passo do ciclo, consiste no incremento do respectivo valor:

r=r+1

Para completar a descrição do ciclo resta considerar a *acção* de (eventual) alteração do estado da máquina URM. Nesta secção do ciclo há que identificar a instrução URM a executar de seguida, extrair o(s) respectivo(s) argumento(s) e agir em conformidade. Podemos recorrer aqui à estrutura *Switch.*

O número da instrução URM a executar é apontado pelo valor da variável *k*. Assim essa instrução será dada por *p[[k]]* e pode ser identificada com o auxílio da função *Mathematica Head*. A expressão que surge como primeiro argumento da construção *Switch* pode então escrever-se como *Head[p[[k]]]*.

São apresentados de seguida as sequências correspondentes à execução de cada uma das instruções URM:

< Executa Zero >

Aqui temos de identificar apenas qual o número do registo cujo valor deve ser igualado a zero. Para aceder a esse número podemos aplicar a expressão *p[[k,1]]*. Alternativamente também poderíamos fazer *p[[k]][[1]]* ou ainda *Part[p[[k]],1]*. Há então que igualar o registo *p[[k,1]]* a zero e avançar para a instrução seguinte incrementando o valor de *k*.

m[p[[k,1]]]=0; k=k+1

< Executa Sucessor >

Aqui temos de identificar qual o número do registo que queremos incrementar, tomar o respectivo conteúdo, incrementá-lo e deixar o resultado nesse mesmo registo. Por fim deve avançar-se para a instrução seguinte incrementando o valor de *k*.

m[p[[k,1]]] = m[p[[k,1]]]+1; k=k+1

< Executa Transferência >

Neste caso existem dois argumentos: o primeiro indica o registo origem e o segundo o registo destino. O objectivo é tomar o valor do registo origem (sem o alterar) e atribui-lo ao registo destino (esquecendo o valor anterior deste). Por fim há que avançar para a instrução seguinte incrementando a variável k.

m[p[[k,2]]]=m[p[[k,1]]]; k=k+1

< Executa Salto >

Neste caso temos três argumentos. Os dois primeiros referem-se aos números dos registos cujos valores devem ser comparados. Se o resultado da comparação for verdadeiro então devemos saltar para a instrução indicada pelo terceiro argumento, atribuindo o valor deste à variável *k*, senão devemos continuar para a instrução seguinte incrementando o valor de *k*. A instrução de salto pode ser simulada por meio de um *If*:

If [m[p[[k,1]]]==m[p[[k,2]]]], k=p[[k,3]], k=k+1]

Juntando tudo o que foi dito anteriormente podemos apresentar a instrução *Switch* na sua globalidade:

```
Switch[Head[p[[k]]],
        Z, m[p[[k,1]]] = 0; k = k + 1,
        S, m[p[[k,1]]] = m[p[[k,1]]] + 1; k = k + 1,
        T, m[p[[k,2]]] = m[p[[k,1]]]; k = k + 1,
        J, If[m[p[[k,1]]] == m[p[[k,2]]], k = p[[k,3]], k = k + 1]
 ];
```

3.7.5 Devolução do resultado

Depois de terminada a execução do ciclo *While* resta devolver o resultado que será:

- o valor de *m[1]* se o programa URM foi executado até ao fim.

ou então:

- uma mensagem a dizer que o programa não terminou ao fim do número de passos estabelecido.

Sabemos que o programa URM terminou se no final for *k>s* onde s é o número de instruções do programa. Assim a expressão que devolve o resultado do programa *executaURM* pode ser escrita, com a ajuda de uma instrução *If*, como se segue:

```
If [k>s, m[1], Print["O programa não terminou ao fim de ", i, " passos"]]
```

3.7.6 O programa *executaURM* completo

É indicado de seguida o programa *executaURM* completo. Note-se que foi incluida a função *Module* por forma a encapsular as variáveis *argList*, *n*, *r* e *k* bem como a função *m*. Evitam-se assim quaisquer efeitos colaterais sobre outras variáveis ou funções que porventura tenham o mesmo nome.

```
executaURM[p_, args__, i_] :=

Module[{argsList, n, r, m, k, s},


(* ----CONFIGURAÇÃO INICIAL--------------------*)

argsList = List[args];
n = Length[argsList];
r = 1;

While[r <= n,
    m[r] = argsList[[r]];
    r = r + 1
];

m[_] = 0;


(* ----EXECUÇÃO DO PROGRAMA URM----------------*)

s = Length[p];
k = 1;
r = 0;

While[k <= s && r <= i,

        Switch[Head[p[[k]]],

             Z, m[p[[k, 1]]] = 0; k = k + 1,

             S, m[p[[k, 1]]] = m[p[[k, 1]]] + 1; k = k + 1,

             T, m[p[[k, 2]]] = m[p[[k, 1]]]; k = k + 1,

             J, If[m[p[[k, 1]]] == m[p[[k, 2]]], k = p[[k, 3]], k = k + 1]

        ];

        r = r + 1
];


(* ----RESULTADO--------------------------------*)

If[k > s, m[1], Print["O programa não terminou ao fim de ", i, " passos"]]

];
```

3.8 Utilizando o programa

3.8.1 Escolha do número de passos a efectuar

A parte final da aula é dedicada à utilização do programa *executaURM*. Antes porém importa discutir o problema da escolha do numero máximo de passos a considerar na execução de um determinado programa URM.

Note-se que o valor de *i* deve ser escolhido com algum cuidado, tendo em conta não só o programa URM em execução mas também a amplitude dos próprios argumentos. Por exemplo, o programa URM "maior" termina sempre (pois dados dois naturais é sempre possível dizer qual dos dois é o maior em sentido lato). No entanto o número de passos decorridos até a terminação do programa depende dos valores dos argumentos como se pretende ilustrar com a tabela seguinte:

x	**y**	**nº passos até a terminação do programa**
0	0	1
5	0	1
0	5	3
7	8	31
10	1	5
1	10	7
260	150	601

Por exemplo:

```
executaURM[maior,7,8,100]
```

termina ao fim 31 passos (<100) obtendo-se a resposta esperada:

```
8
```

mas, por exemplo:

```
executaURM[maior,260,150,100]
```

devolve a mensagem:

O programa não terminou ao fim de 100 passos.

Neste caso esta mensagem significa apenas que o programa não terminou ao fim de 100 passos o que não quer dizer que não termine ao fim de um número de passos superior. Neste caso concreto o programa só termina ao fim de 601 passos. Assim se tivéssemos feito por exemplo:

executaURM[maior,260,150,900]

obteríamos a resposta desejada:

260

Sempre que existam dúvidas, sobre a terminação do programa, devemos aumentar o valor do parâmetro *i* e tentar de novo.

3.8.2 Exemplos

Vamos aqui considerar a aplicação do programa *executaURM* para testar o funcionamento de alguns dos programas URM desenvolvidos ou propostos nas aulas Teórico-Práticas.

<u>Exemplo 1</u> (Construído na 1ª aula Teórico-Prática):

$f : IN_o \rightarrow IN_o$ com $f(x) = 5 \quad \forall x \in IN_o$

I_1 Z(1)
I_2 S(1)
I_3 S(1)

I_4 S(1)

I_5 S(1)

I_6 S(1)

```
f={Z[1], S[1], S[1], S[1], S[1], S[1]};
```

```
executaURM[f, 10, 20]
```

```
5
```

```
executaURM[f, 565245676657, 20]
```

```
5
```

Exemplo 2 (Construído na 1ª aula Teórico-Prática):

$f : IN_o \rightarrow IN_o$ com $f(x) = x \text{ Div } 2$ (divisão inteira de x por 2)

I_1 J(1,2,7)

I_2 S(2)

I_3 J(1,2,7)

I_4 S(3)

I_5 S(2)

I_6 J(1,1,1)

I_7 T(3,1)

```
div2={J[1,2,7], S[2], J[1,2,7], S[3], S[2], J[1,1,1], T[3,1]};
```

```
executaURM[div2, 20, 100]
```

```
10
```

```
executaURM[div2, 25, 100]
```

```
12
```

Exemplo 3 (Sugerido como trabalho de casa na 1ª aula Teórico-Prática):

$f : IN_o \to IN_o$ com $f(x) = x \text{ Mod } 2$ (resto da divisão inteira de x por 2)

I_1 J(1,2,9)
I_2 S(2)
I_3 J(1,2,6)
I_4 S(2)
I_5 J(1,1,1)
I_6 Z(1)
I_7 S(1)
I_8 J(1,1,10)
I_9 Z(1)

```
mod2={J[1,2,9], S[2], J[1,2,6], S[2], J[1,1,1], Z[1], S[1], J[1,1,10], Z[1]};
```

```
executaURM[mod2, 20, 100]
```

```
0
```

```
executaURM[mod2, 25, 100]
```

```
1
```

Exemplo 4 (a construir nesta aula):

$$g : IN_o \to IN_o \text{ tal que } g(x) = \begin{cases} 1 & , \ x = 0 \\ x - 2 & , \ x \text{ par} \neq 0 \\ x + 2 & , \ x \text{ impar} \end{cases}$$

I_1 J(1,2,16)
I_2 S(2)
I_3 J(1,2,15)
I_4 S(2)
I_5 J(1,2,7)

I_6 J(1,1,2)

I_7 S(3)

I_8 S(3)

I_9 J(1,3,13)

I_{10} S(4)

I_{11} S(3)

I_{12} J(1,1,9)

I_{13} T(4,1)

I_{14} J(1,1,17)

I_{15} S(1)

I_{16} S(1)

```
g={J[1,2,16], S[2], J[1,2,15], S[2], J[1,2,7], J[1,1,2], S[3], S[3], J[1,3,13],
S[4], S[3], J[1,1,9], T[4,1], J[1,1,17], S[1],S[1]};
```

```
executaURM[g, 0, 20]
```

```
1
```

```
executaURM[g, 5, 20]
```

```
7
```

```
executaURM[g, 10, 20]
```

```
O programa nao terminou ao fim de 20 passos
```

```
executaURM[g, 10, 100]
```

```
8
```

Exemplo5 (trabalho para casa):

$h : IN_o^3 \to IN_o$ tal que $h(x, y, z) = x + y + z$

I_1 J(2,4,5)

I_2 S(1)

I_3 S(4)

I_4 J(1,1,1)

I_5 Z(4)

I_6 J(3,4,10)

I_7 S(1)

I_8 S(4)

I_9 J(1,1,6)

```
h={J[2,4,5], S[1], S[4], J[1,1,1], Z[4], J[3,4,10], S[1], S[4], J[1,1,6]};
```

```
executaURM[h, 1, 2, 3, 50]
```

```
6
```

```
executaURM[h, 0, 2, 3, 50]
```

```
5
```

```
executaURM[h, 5, 6, 0, 50]
```

```
11
```

4 Conclusão

Pretendeu-se com esta lição mostrar que é perfeitamente possível construir com os alunos, no decurso de uma aula prática, recorrendo ao *Mathematica*, o emulador da máquina URM.

A introdução desta aula no plano da cadeira de Teoria da Computabilidade e Complexidade já foi acordada com o Professor Doutor Eduardo Fermé, actual responsável pela cadeira, pelo que será uma realidade já no próximo ano lectivo.

Ao frequentarem a aula os alunos acabarão por interiorizar melhor a estrutura e programação da máquina URM. Espera-se, assim, assistir nas aulas seguintes, ainda dedicadas à construção de programas URM, a uma maior produtividade.

Com este exemplo concreto pretendeu-se também relembrar aos alunos que o *Mathematica* é uma excelente ferramenta de trabalho à qual podem recorrer para os auxiliar na resolução de problemas de outras cadeiras, assim como aumentar a prática de programação dos alunos através de um exemplo que envolve a combinação de vários paradigmas da programação (nomeadamente a programação imperativa e a reescrita e manipulação simbólica de expressões).

Anexo A

A.1 Codificação de Programas URM

Definição:

> *Um conjunto X diz-se enumerável se e só se existir uma bijecção* $f : X \rightarrow \mathbb{N}_0$.

- $\mathbb{N}_0 \times \mathbb{N}_0$ é enumerável.

$$\pi : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0 \quad \text{com} \quad \pi(m,n) = 2^m(2n+1)-1$$

- $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ é enumerável

$$\zeta : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}_0 \quad \text{com} \quad \zeta(m,n,q) = \pi[\pi(m-1,n-1),q-1]$$

- O conjunto formado por todas as sequências finitas de números naturais é enumerável.

$$\tau : \cup_{k>0} \mathbb{N}^k \rightarrow \mathbb{N}_0$$
$$\text{com} \quad \tau(a_1,\ldots,a_k) = 2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} + \ldots + 2^{a_1+a_2+a_3+\ldots+a_k+k-1} - 1$$

- O conjunto $\mathcal{I}$ composto por todas as instruções URM é enumerável.

$$\beta : \mathcal{I} \rightarrow \mathbb{N}_0 \quad \text{com} \quad \beta[Z(n)] = 4(n-1);$$
$$\beta[S(n)] = 4(n-1)+1;$$
$$\beta[T(m,n)] = 4\pi(m-1,n-1)+2;$$
$$\beta[J(m,n,q)] = 4\zeta(m,n,q)+3;$$

- O conjunto $\mathcal{P}$ composto por todas os programas URM é enumerável.

$$\gamma : \mathcal{P} \rightarrow \mathbb{N}_0 \quad \text{com} \quad \gamma(P) = \tau[\beta(I_1), \beta(I_2), \ldots, \beta(I_k)]$$

(onde P é um programa URM)

Assim, para codificar um programa URM, P, basta calcular a função $\gamma(P)$.
Implementação das funções π, ζ, τ, β e γ no *Mathematica*:

```
(* FUNCAO PI *)

pi=Function[{m,n},2^m*(2n+1)-1];
```

```
(* FUNCAO ZETA *)

zeta=Function[{m,n,q},pi[pi[m-1,n-1],q-1]];

(* FUNCAO TAU *)

tau=Function[w,
    If[Length[w]= =1,2^w[[1]]-1,
     2^(Apply[Plus,w]+Length[w]-1)+tau[Drop[w,-1]]]];

(* FUNCAO BETA *)

betaz=Function[n,4(n-1)];
betas=Function[n,4(n-1)+1];
betat=Function[{m,n},4*pi[m-1,n-1]+2];
betaj=Function[{m,n,q},4*zeta[m,n,q]+3];
beta=Function[i,Switch[Head[i],
     z,betaz[i[[1]]],
     s,betas[i[[1]]],
     t,betat[i[[1]],i[[2]]],
     j,betaj[i[[1]],i[[2]],i[[3]]]
     ]
    ];

(* FUNCAO GAMA *)

gama=Function[p,tau[Map[beta,p]]];
```

<u>Exemplos</u>:

```
gama[{s[2], j[1, 1, 2], z[1], s [4], t[4, 1]}]
```

```
9223372041150136351
```

```
maior = {j[2, 3, 6], j[1, 3, 5], s[3], j[1, 1, 1], t[2, 1]}; gama[maior]
```

```
105005015402898283611219023743032286083285989880497170097198954499999945331531\
135971481846797516413432297887324754770457139961860283561621972569580008480567\
059991319509996847067998728362574716399669406336928158586421745546308776137560\
088759230344783570654084596694195374266552565243586550620852807147432659079587\
406239717901823028661664085726558729965582616584082940121509670186031231575985\
086728532195445808070783719995508481792861148344607432133468419959913910100933\
359841369937762693842208224035614170130064091071396657785229004719728534425426\
477297678361216267015119931348455844462800882205709009759828494425441612910522\
689858416364693666864389364058785759460796807671205353476228659556837455356996\
682130980486500498411823284293290973503459426187601875901491119788275357697057\
620875316615140916480979239603263603369369078228205505536495990520263927646166\
321481000088983737181105792730108393410793110274477537917056955448405636582435\
282284805410185385676687054098247729509873463087001678960831434015211304471957\
```

```
80942523220834962869610867311318952848472782088878889130332542899153392652038 4\
```

```
8625074292387730365213681050648745401456867499543858006963849559915821424237 68\
40056536223128269692594788539946568028863144400635729071429737739879012940053 1\
04796055773416427444936201334981264213586022001634190930655549858038502711432 6\
41256818507066464147929841145916320734692728732696428097571571045631926791660 0\
46048140778168028593504239530083166145663701768862954542214715570999685758595 1\
49967060442730327392425780577395482996857108474605265167797221820470975936903 7\
77984999245471245348299939095752416248522747932386575333761430115863771922396 1\
74175360408380736033428009258495780647854263014931062773023571506096886945399 1\
03006646951875299603603573776919287897481051013317659300353915975317451386725 7\
17466727824028231996123512798112976167965467628890203766496681350195038813564 2\
27646478000167953759105043470296641938062148202024776610378193917837638051380 0\
35515619799012607882715424121069720152169427084365447308637616191855632271411 5\
72912155820023078451877146300233906142199024556459156234667080863232307460603 4\
86238729299576005826097698548465710073998434593006420011440591993277618026091 4\
90078324008051232463328398893472026027921308309819547238684717727913572093635 3\
44732168698767993854305293532822388842143139726466267857005605704372432711063 0\
6508879735817869721 5
```

A.2 Descodificação de Programas URM

Qualquer número natural representa a codificação de um programa URM. Por exemplo, para obter o programa URM cujo código é *n*, é necessário calcular o resultado de $\gamma^{-1}(P)$. Para isso é preciso conhecer também as inversas das funções π, ζ, τ e β.

- $\pi^{-1} : \mathbb{N}_0 \rightarrow \mathbb{N}_0 \times \mathbb{N}_0$ com $\pi^{-1}(x) = (\pi_1(x), \pi_2(x))$

  onde:

  $\pi_1(x) = (x+1)_1$ é o expoente do primeiro número primo da factorização de x+1 em factores primos.

  e:

  $$\pi_2(x) = \frac{1}{2}\left[\frac{x+1}{2^{\pi_1(x)}} - 1\right]$$

- $\zeta^{-1} : \mathbb{N}_0 \rightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ com $\zeta^{-1}(x) = [\pi_1(\pi_1(x))+1, \pi_2(\pi_1(x))+1, \pi_2(x)+1]$

- $\tau^{-1} : \mathbb{N}_0 \rightarrow \cup_{k>0} \mathbb{N}^k$ com $\tau^{-1}(x) = (a_1, a_2, \ldots, a_k)$

  onde:

  $$x + 1 = 2^{b_1} + 2^{b_2} + 2^{b_3} + \ldots + 2^{b_k}$$

  e:

$$\begin{cases} a_1 = b_1 \\ a_2 = b_2 - b_1 - 1 \\ a_3 = b_3 - b_2 - 1 \\ \ldots \\ a_k = b_k - b_{k-1} - 1 \end{cases}$$

- $\beta^{-1} : \mathbb{N}_0 \to \mathcal{I}$

  Considerando x = 4u+r (com u e r naturais) fica:

  $r = 0 \Rightarrow \beta^{-1}(x) = z(u+1)$

  $r = 1 \Rightarrow \beta^{-1}(x) = s(u+1)$

  $r = 2 \Rightarrow \beta^{-1}(x) = T(\pi_1(u)+1, \pi_2(u)+1)$

  $r = 3 \Rightarrow \beta^{-1}(x) = J(m,n,q)$ com $(m,n,q) = \zeta^{-1}(u)$

- $\gamma^{-1} : \mathbb{N}_0 \to \mathcal{P}$ com $\gamma^{-1}(x) = (\beta^{-1}(a_1), \beta^{-1}(a_2),..., \beta^{-1}(a_k))$

  onde:

  $(a_1,a_2,...,a_k) = \tau^{-1}(x)$

Implementação das inversas das funções π, ζ, τ, β e γ no *Mathematica*:

```
(* IVN FUNCAO PI *)

pi1 = Function[x, If[EvenQ[x], 0, Flatten[FactorInteger[x + 1]][[2]]]];
pi2 = Function[x, 1/2*((x + 1)/(2^(pi1[x])) - 1)];

(* IVN FUNCAO ZETA *)

zeta1 = Function[x, pi1[pi1[x]] + 1];
zeta2 = Function[x, pi2[pi1[x]] + 1];
zeta3 = Function[x, pi2[x] + 1];
invzeta = Function[x, {zeta1[x], zeta2[x], zeta3[x]}];

(* IVN FUNCAO TAU *)

invtau = Function[x, res = {}; d = x + 1;
    While[d >= 1, res = Append[res, Mod[d, 2]]; d = Quotient[d, 2];];
    i = 1; res1 = {};
```

```
    While[i <= Length[res],
      If[res[[i]] != 0, res1 = Append[res1, res[[i]]*i - 1]]; i = i + 1];
    sol = {First[res1]};
    i = 1;
    While[i < Length[res1],
      sol = Append[sol, res1[[i + 1]] - res1[[i]] - 1]; i = i + 1]; sol
    ];

(* IVN FUNCAO BETA *)

invbeta = Function[x,
    u = Quotient[x, 4]; r = Mod[x, 4];
    Switch[r,
      0, z[u + 1],
      1, s[u + 1],
      2, t[pi1[u] + 1, pi2[u] + 1],
      3, k = invzeta[u]; j[k[[1]], k[[2]], k[[3]]]
      ]
    ];

(* IVN FUNCAO GAMA *)

invgama = Function[x, Map[invbeta, invtau[x]]];
```

Exemplos:

```
invgama[9223372041150136351]
```

```
{s[2], j[1, 1, 2], z[1], s[4], t[4, 1]}
```

```
invgama[gama[maior]]
```

```
{j[2, 3, 6], j[1, 3, 5], s[3], j[1, 1, 1], t[2, 1]}
```

Bibliografia

J. Carmo, A. Sernadas, C. Sernadas, F. M. Dionísio e C. Caleiro, *Introdução à Programação em Mathematica*, IST Press, 1999

Cultland N. *Computability - An introduction to recursive function theory*, Cambridge University Press, 1980

Wolfram S., *The Mathematica book*, Wolfram Media, 3ª edição, 1996