GUIA DE UTILIZAÇÃO DO AXIOM

S. C. COUTINHO

Resumo. Nestas notas descrevemos como usar o Axiom para fazer cálculos e escrever programas simples relativos ao curso de *Números inteiros e criptografia*, baseado no livro [4]. Estas notas pressupõem que você está utilizando o Axiom no Windows, a partir do `command prompt`. É possível também usá-lo a partir do TeXmacs e do SAGE, mas não trateremos disto aqui.

Sumário

Date: 3 de setembro de 2007.

1. Introdução

Para entrar no Axiom clique em `Iniciar | Programas | Axiom | Axiom`. Com isto você vai abrir uma tela como a da figura 1.

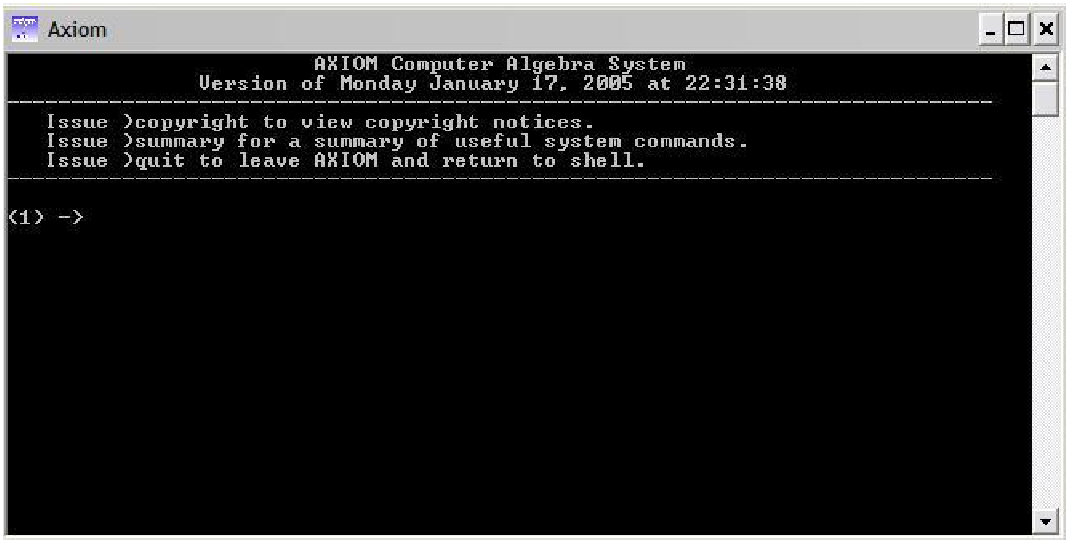

Figura 1. Tela inicial

Para sair do Axiom escreva `)quit` ou simplesmente `)q`, seguido de `enter`. Como todas as entradas devem ser seguidas de `enter` para que sejam processadas pelo sistema, vamos omitir a menção a `enter` de agora em diante. O sistema vai perguntar se você deseja mesmo sair: `y` sai completamente e `n` retorna você ao sistema.

Observe que, os comandos que você aprendeu até agora começam todos com um parêntesis `)`. Não é verdade que todos os comandos do Axiom têm esta forma. Na verdade, os comandos que começam com `)` são apenas os chamados *comandos de sistema* (*System Commands*) que permitem a você controlar o ambiente do sistema. Há muitos destes comandos e trataremos aqui apenas de uns poucos.

O primeiro destes é `)clear all` que é usado para limpar toda a área de trabalho, de modo que, por exemplo, nomes de variáveis e nomes de funções possam ser reutilizados sem interferência do que tenha sido feito antes. Outro comando muito útil é `)show`, que pode ser usado para mostrar informação sobre um domínio ("domain"), pacote ("package") ou categoria ("category") do Axiom. O significado exato destes termos será explicado ao longo dos laboratórios. Por exemplo, para saber tudo sobre inteiros, digite

```
)show Integer
```

Por outro lado, se você quer apenas as operações permitidas com inteiros no AXIOM, ou somente os atributos de `Integer` digite

```
)show Integer )operations
```

ou

```
)show Integer )attributes
```

A propósito, `Integer` pode ser abreviado como `INT`. Note o uso de maiúsculas e minúsculas! Já `)what` é usado para gerar listas: de operações, categorias, pacotes, etc. Por exemplo,

```
)what operations INT
```

gera uma lista de todas as operações que contêm o padrão `int` em seu nome. Para obter informações sobre uma operação específica, usamos `)display`. Por exemplo,

```
)display op gcd
```

retorna informação sobre o máximo divisor comum ("greatest common divisor") de dois elementos que podem pertencer a várias categorias diferentes do AXIOM, entre as quais estão inteiros não negativos e inteiros positivos.

Como o sistema será usado em algumas de nossas provas, precisamos ter uma maneira de gerar um arquivo no qual o sistema escreve tudo o que você for fazendo. Para isto usamos `)spool`. Por exemplo

```
)spool c:/collier/programas/axiom/lab1
```

faz com que o sistema escreva tudo o que estou fazendo num arquivo chamado `lab1.output` que está na pasta `c:/collier/programas/axiom/` do meu computador. Para fazer o sistema parar de escrever neste arquivo para usamos `)spool` sem escrever o nome de nenhum arquivo em seguida.

Encerramos com dois comandos de sistema que serão usados mais freqüentemente quando começarmos a programar no AXIOM. Os comandos são `)cd` para mudar de diretório e `)read` para ler um programa. Por exemplo,

```
)cd c:/collier/programas/axiom/
)read prova1
```

faz com que o sistema passe a tomar `c:/collier/programas/axiom/` como seu diretório "default" e em seguida leia o arquivo `prova1.input` que está nesta pasta. A propósito, para um arquivo ser executado pelo AXIOM usando `)read` ele deve ter terminação `.input`.

2. Tipos

O Axiom é um sistema em que cada objeto tem um tipo. O tipo de um objeto é determinado pelo *domínio de computação* ao qual ele pertence, muitos dos quais correspondem a uma estrutura matemática bem determinada. Exemplos de domínios que usaremos neste curso e suas abreviações são:

Domínio de computação	Abreviação	Descrição
`Integer`	`INT`	inteiros
`PositiveInteger`	`PI`	inteiros positivos
`Float`	`FLOAT`	números reais em ponto flutuante
`Fraction(Integer)`	`FRAC`	frações
`UnivariatePolynomial`	`UP`	polinômios em uma variável

Tabela 1. Alguns tipos

Ao dar entrada a um objeto, *não é necessário* dizer ao sistema qual é seu tipo. O próprio sistema vai procurar determinar qual o tipo mais adequado. Isto é absolutamente necessário, porque o domínio ao qual um objeto pertence (isto é, seu tipo) determina quais são as operações que podem ser aplicadas àquele objeto. Por exemplo, dando entrada a 2, obtemos

```
(1) -> 2

   (1) 2

                                                    Type: PositiveInteger
```

Note que nada do que fizemos antes disto alterou a numeração do `prompt`, mas ao escrever 2, a numeração foi incrementada. Isto é, comandos de sistema não são contados como linhas dos cálculos que você está fazendo. Outra observação: ao contrário de quase todos os outros sistemas, as linhas do Axiom *não precisam* acabar em ponto-e-vírgula. Não precisam, mas podem acabar. Entretanto, se você adicionar o ponto-e-vírgula ao final, o sistema não mostra, como acima, o objeto que recebeu; embora indique seu tipo. Por exemplo:

```
(2) -> 2;

                                                    Type: PositiveInteger
```

O fato de cada objeto ter um tipo determinado pode causar algumas surpresas e mesmo fazer o programa não efetuar o cálculo desejado. Antes de dar um exemplo, vejamos como atribuir um valor a uma variável. Para isto usamos o operador de atribuição :=. Por exemplo,

```
(3) -> x:= 2

   (3) 2

                                                    Type:  PositiveInteger
```

Contudo, se quisermos que o sistema considere este 2 como uma fração, devemos determinar que a variável escolhida tem tipo `FRAC(INT)`. Para isto, escrevemos

```
(4) -> y:FRAC(INT):= 2

   (4) 2

                                                    Type:  Fraction Integer
```

Calculando agora $\mathrm{mdc}(x, x)$ e $\mathrm{mdc}(y, y)$, vemos a diferença entre os tipos produzir resultados bem distintos:

```
(5) -> gcd(x, x)

   (5) 2

                                                    Type:  PositiveInteger

(6) -> gcd(y,y)

   (6) 1

                                                    Type:  Fraction Integer
```

Exercícios 2.1. *Explique porque o* Axiom *retorna* 1 *como sendo o máximo divisor comum de* 2 *quando considerado como um elemento de* $\mathbb{Q}$.

Para nossa sorte, o Axiom converte alguns tipos em outros automaticamente. Por exemplo,

```
(7) -> 1/x

   (7)  1/2

                                                    Type:  Fraction Integer
```

Contudo, algumas destas conversões podem surpreendê-lo. Por exemplo,

```
(8) -> sqrt(x)

   (8)  √2

                                                    Type:  Algebraic Number
```

Número algébrico? Que espécie de tipo é este? Em álgebra um número complexo é *algébrico* se for solução de alguma equação com coeficientes racionais. Por exemplo, qualquer número da forma

$$a + b\sqrt{d} \quad \text{onde} \quad a, b \in \mathbb{Z}$$

é algébrico. De fato, escrevendo $\alpha = a + b\sqrt{d}$, temos que

$$(\alpha - a)^2 = (b\sqrt{d})^2 = b^2 d;$$

de modo que α é raiz da equação polinomial com coeficientes inteiros

$$f(x) = (x - a)^2 - bd^2;$$

o que nos permite concluir que α é algébrico. Na verdade, podemos efetuar todos os cálculos acima usando o próprio AXIOM. Para isto, começamos definindo α:

```
(9) -> alpha:= a+b*sqrt(d)

   (9) b sqrt(d) + a

                                                    Type:  Expression Integer
```

Observe que obtivemos um novo tipo, chamado *expressão inteira* (abreviado como `EXPR`). Para saber mais sobre este tipo você pode usar `)show EXPR`, se quiser. Voltando aos cálculos, queremos elevar $\alpha - a$ ao quadrado:

```
(10) -> (alpha-a)^2

   (10) b^2d

                                                    Type:  Expression Integer
```

Para definir a equação polinomial fazemos

```
(11) -> f:=(x-a)^2-b*d^2

   (11) x^2-2ax-b^2d+a^2

                                                    Type:  Polynomial Integer
```

Que produz um novo tipo, chamado polinômio inteiro; isto é, f é um polinômio com coeficientes inteiros. Para testar se α é mesmo raiz de f, substituímos x por α em f:

```
(12) -> eval(f,x=alpha)

   (12) 0

                                                    Type:  Expression Integer
```

que é uma expressão do mesmo tipo de α, naturalmente. Nos cálculos acima usamos, sem maiores cerimônias, vários símbolos para operações. Uma lista destes e de mais alguns nomes de operações pode ser encontrado na tabela 2.

Operação	Símbolo usado no AXIOM
Adição	+
Subtração	-
Multiplicação	*
Divisão	/
Raiz quadrada	`sqrt`
Divisão com resto	`divide`
Quociente de uma divisão	`quo`
Resto de uma divisão	`rem`
Exponenciação	^ ou **

TABELA 2. Algumas operações elementares

Note que `divide`, `quo` e `rem` recebem dois argumentos: os dois números aos quais vai ser aplicada a divisão com resto. Por exemplo,

```
(13) -> divide(12,5)

   (13) [quotient = 2, remainder=2]
                                Type: Record(quotient: Integer, remainder: Integer)
```

Mais uma vez, acabamos com um novo tipo; desta vez chamado de "Record". Este é um tipo específico do AXIOM e permite formar listas de objetos não homogêneos. Não precisamos discutir este tipo em detalhes, mas é conveniente aprender a ter acesso aos campos do `Record` porque isto será necessário na próxima seção. Para isto daremos um nome à nossa divisão:

```
(14) -> L:= divide(12,5)

   (14) [quotient = 2, remainder=2]
                                Type: Record(quotient: Integer, remainder: Integer)
```

Para obter o valor do quociente basta fazer

```
(15) -> L.quotient

   (15) 2
                                                          Type: PositiveInteger
```

o resto é obtido através de `L.remainder`

O que fizemos até agora sobre tipos é suficiente para os cálculos mais substanciais com inteiros da próxima seção.

3. Inteiros

Começamos com uma tabela que contém mais algumas operações para inteiros disponíveis no Axiom.

Operação	Descrição
`gcd(m,n)`	máximo divisor comum de m e n
`lcm(m,n)`	mínimo múltiplo comum
`extendedEuclidean(m,n)`	algoritmo euclidiano estendido aplicado a m e n
`factorial(n)`	calcula o fatorial de n
`positive?(n)`	retorna `True` se $n > 0$, se não retorna `False`
`prime?(n)`	retorna `True` se n for primo, se não retorna `False`
`nextPrime(n)`	primo seguinte a n

Tabela 3. Mais operações com inteiros

É importante notar que `extendedEuclidean(m,n)` retorna um `Record` com três entradas. Se escrevemos d para o máximo divisor comum de m e n e

$$d = \alpha m + \beta n$$

pelo algoritmo estendido, então o `Record` é

`[coef1=` α`,coef2=` β`,generator=` d`]`.

Como já sabemos acessar cada entrada de um `Record`, fica fácil resolver a questão abaixo usando o Axiom. Para isto siga o roteiro no exercício.

Exercícios 3.1. *Alberto costumava receber R$ 1239,00 de salário, mas teve um aumento, e seu salário passou para R$ 1455,00. Para uma prestação de contas ele precisa saber o número m de meses durante os quais recebeu o salário menor e o número n de meses durante os quais recebeu o maior. A única coisa que Alberto sabe é que recebeu um total de R$ 21786,00 no período da prestação de contas.*

Para resolver a questão precisamos encontrar soluções inteiras para a equação diofantina

$$1239x + 1455y = 21786.$$

Para isto siga as seguintes etapas:

(1) *Escreva a equação* $1239x + 1455y = 21786$*, atribuindo-lhe um nome, digamos E. Qual o seu tipo?*
(2) *Calcule o máximo divisor comum de m e n.*

(3) *Divida a equação pelo máximo divisor comum e determine se pode ter solução (inteira, claro!).*
(4) *Aplique o algoritmo euclidiano estendido a m e n, atribuindo um nome ao resultado para que possa acessá-lo mais tarde.*
(5) *Calcule a solução particular de E.*
(6) *Escreva fórmulas para os valores de x e y nas soluções gerais de E atribuindo-lhes nomes; por exemplo,* `x_0` *e* `y_0`*.*
(7) *Use* `eval` *para testar se sua solução geral está correta. Cuidado: você deve substituir uma variável de cada vez.*
(8) *O* AXIOM *tem uma função já pronta para resolver equações diofantinas. Use* `)what op diophantine` *para descobri-la e* `)display op` *para descobrir como é sua sintaxe.*

Nosso segundo exercício é semelhante ao primeiro exercício da primeira prova. Como se trata de uma demonstração por indução, seria bom escrever comentários explicando o que está sendo feito em cada etapa. Você pode fazer isto no AXIOM colocando os comentários entre aspas:

```
(16) -> "A base da inducao consiste em tomar n =1, neste caso"

   (16) "A base da inducao consiste em tomar n =1, neste caso"

                                                        Type: String
```

Note que as aspas transformaram a frase em uma `string` que o sistema simplesmente repete. Para evitar a repetição destas linhas no arquivo `output` e tornar sua demonstração mais legível, basta pôr o ponto-e-vírgula no final da linha. A descrição do procedimento não inclui os comentários, que ficam por sua conta.

Exercícios 3.2. *O objetivo deste exercício é provar, por indução em n, que $n^{11} - n$ é múltiplo de 11 qualquer que seja o inteiro $n \geq 1$.*

(1) *Seja $E := n^{11} - n$.*
(2) *Mostre que o resultado vale quando $n = 1$.*
(3) *Calcule R tal que*
$$(n+1)^{11} - (n+1) = n^{11} - n + R.$$
(4) *Mostre que R é divisível por 11. Há várias maneiras de fazer isto. Por exemplo, você pode dividir R por 11 e verificar que o resto dá inteiro; também pode usar* `factor` *para fatorar R e fazer aparecer um 11 em evidência.*
(5) *Conclua o resultado desejado.*

Vimos que $n^k - n$ é divisível por k quando $k = 11$. Vejamos, agora, um exemplo em que o resultado é falso. Para isto, basta escolher o k e calcular $n^k - n$ para alguns

valores de n, até encontrar um que não seja múltiplo de k. Vamos usar o AXIOM para gerar o conjunto $n^k - n$ para $k = 0, \ldots, k-1$ e ver o que acontece. Ah, sim, quase ia esquecendo: precisamos escolher um valor para k. Que tal 12? Para gerar o conjunto desejado, basta fazer

(17) -> `S:= set[n^(12)-n for n in 0..11]`

Note que o resultado tem tipo `Set Integer`. Infelizmente ficou meio difícil decidir se estes números são, ou não, múltiplos de 12. Por isso é melhor calcular o conjunto dos restos da divisão de $n^{12} - n$ por 12, usando a função `rem`. Se algum dos $n^{12} - n$ não for múltiplo de 12, obteremos números diferentes de 0 no conjunto.

(18) -> `S:= set[rem(n^(12)-n),12) for n in 0..11]`

Qual sua conclusão? Seria uma boa idéia testar isto com outros valores de k. Para não ter que reescrever a definição de S cada vez que escolhermos um novo valor para k, podemos definir uma função $S(k)$ que calcula o conjunto quando atribuímos um valor específico para k. Fazemos isto escrevendo:

(19) -> `S(k) == set[rem(n^(k)-n),k) for n in 0..k-1]`

Note que, para definir uma função usamos um novo tipo de atribuição, denotado por ==, em vez do := usual.

Exercícios 3.3. *Para que valores de $2 \leq k \leq 20$ o conjunto $S(k)$ só contém zero? Se não quiser ir calculando os conjuntos um a um, você pode gerar um conjunto cujos elementos são os conjuntos $S(k)$.*

A grande pergunta é: olhando para os resultados numéricos obtidos no exercício acima, o que você conclui sobre os valores de k para os quais $n^k - n$ é múltiplo de k?

Exercícios 3.4. *Para pôr sua conclusão à prova, escolha um valor adequado de k entre 50 e 100, e mostre por indução em n que $n^k - n$ é divisível por k para o valor que você escolheu.*

4. Laços

Nesta seção veremos como utilizar os vários tipos de laços disponíveis no AXIOM. Afinal, sem eles, não podemos escrever nenhum programa interessante. Contudo, para usar os laços, não podemos continuar utilizando o AXIOM em modo interativo. A partir de agora devemos escrever um pequeno programa e mandar o AXIOM executá-lo.

Para começar, precisamos de um editor de texto com o qual o programa será escrito. Nestas notas vou supor que o editor utilizado é o NOTEPAD do WINDOWS. Depois de

escrito, o programa deve ser gravado como um arquivo com terminação `.input`. Para fazer isto no NotePad, escreva o nome do arquivo seguido da terminação `.input` entre aspas. Por exemplo, para gravar o arquivo chamado `fermat`, clique em *salvar como*, e na janela correspondente, escreva

```
"fermat.input"
```

no campo *Nome do arquivo*. Para executar este arquivo no Axiom, comece por escolher o diretório onde o arquivo foi gravado como seu *default*, usando o comando `)cd`. Para executar o arquivo basta escrever

```
(1) -> )read fermat
```

Mais detalhes na página 3. Voltando aos laços, o Axiom tem dois tipos básicos: `for` e `while`.

4.1. **Laço `for`.** Este laço tem a forma geral

`-> for` *Variável* in *Intervalo* `repeat`
Instruções

Note que as *Instruções* devem aparecer tabuladas em relação à posição do `for`. Geralmente usamos este laço quando o intervalo de variação da *Variável* (que controla o número de voltas do laço) é conhecido *a priori*. Por exemplo, digamos que queremos calcular o número N_k, definido pela recorrência

$$N_1 = 2 \quad \text{e} \quad N_{k+1} = N_k \cdot (N_k + 1),$$

para $k = 20$. Estes números são utilizados na demonstração de que há infinitos números primos descoberta por Filip Saidak e publicada em 2006 no American Mathematical Monthly (vol. 113). O artigo original pode ser encontado em `http://www.uncg.edu/~f_saidak/`.

Podemos proceder da seguinte maneira:

```
n:=1
for n in 1..100 repeat
    n:=n*(n+1)
n
```

O n que aparece ao final é a saída do programa, e é a única coisa que será exibida na tela, ao final da execução.

Um dos problemas desta maneira de fazer um programa é que só podemos rodá-lo uma vez; ou melhor, para rodá-lo de novo é necessário alterar o arquivo que contém o programa. Veremos como contornar este problema no artigo 5.2.

Enquanto isto, vejamos outro exemplo. Digamos que, desta vez, nosso objetivo seja o de descobrir quais são os inteiros primos, e quais os compostos, entre 2 e 100. Desta vez o programa não deve apenas retornar um valor final: vamos classificar os inteiros um a um à medida que o programa vai rodando. Para descobrir se um inteiro é primo ou composto usamos a função `prime?` do AXIOM. Para fazer com que o programa vá mostrando o resultado à medida que roda, imprimiremos a saída a cada laço executado. Para isto, usamos a função `output`. O programa será o seguinte:

```
n:=2
for n in 1..100 repeat
    output([n,prime?(n)])
```

Este programa imprime na tela pares da forma

[*inteiro positivo*,*valor booleano*]

onde *valor booleano* pode ser `true` ou `false`. Note também que, como a saída vai sendo impressa pelo comando `output` à medida que vai sendo gerada, não há nenhuma variável cujo valor deve ser impresso como saída do programa.

Como o comando `output` pode ser utilizado para fazer o AXIOM retornar valores que estão sendo calculados à medida que o programa é processado, ele é muito útil quando precisamos "debugar" um programa. Para isto podemos fazer o AXIOM imprimir os valores de quantas variáveis desejarmos, à medida que o programa vai sendo executado.

Usaremos freqüentemente o laço `for` na geração de estruturas como conjuntos e listas, como aliás já fizemos na página 10. Trataremos desta questão em mais detalhes no próximo artigo.

4.2. **Listas.** Já vimos como representar conjuntos no AXIOM, mas este é apenas um dos tipos de estrutura de dados presentes neste sistema. Outro tipo, muito útil, são as listas. Ao contrário de um conjunto, os elementos de uma lista estão ordenados. Portanto, embora os conjuntos $\{1, 2\}$ e $\{2, 1\}$ sejam iguais, as listas $[1, 2]$ e $[2, 1]$ são diferentes. Observe que usamos *chaves* para denotar conjuntos e *colchetes* para denotar listas. Mais uma vez, este é um tipo de estrutura que já usamos antes, por exemplo, quando geramos pares em um dos programas do artigo anterior.

Até aqui, uma lista pode parecer muito semelhante a um *array*. Para os nossos propósitos, a principal diferença é que um *array* tem um tamanho fixo e pré-determinado; já a lista, pode ser aumentada ou reduzida conforme as necessidades do programa. Contudo, o AXIOM requer que todos os elementos de uma lista sejam de um mesmo tipo.

Para gerar uma lista com os inteiros 3, 4 e 2 como elementos, basta escrever

```
-> [3,4,2]
```

A tabela 4 resume alguns dos comandos referentes às listas no AXIOM.

Comando	Sintaxe	Descrição
`append`	`append(L_1,L_2)`	acrescenta L_2 ao final de L_1
`cons`	`cons(a,L)`	acrescenta a ao início de L
`first`	`first(L)`	primeiro elemento da lista L
`first`	`first(L,n)`	primeiros n elementos da lista L
`rest`	`rest(L)`	L sem o primeiro elemento
`rest`	`rest(L,n)`	L sem os n primeiros elementos
`reverse`	`reverse(L)`	reverte a ordem de L
`sort`	`sort(L)`	ordena L em ordem crescente
`removeDuplicates`	`removeDuplicates(L)`	retira repetições de L
`#`	`#L`	número de elementos de L
`.`	`L.k`	k-ésimo elemento de L
`.last`	`L.last`	último elemento de L

TABELA 4. Comandos para listas

Podemos gerar listas muito mais complexas embutindo o comando `for` na definição da própria lista. Por exemplo, a lista dos ímpares de 1 a 100 pode ser gerada como

```
-> [2*k+1 for k in 1..49]
```

ou ainda como

```
-> [k for k in 1..100 by 2]
```

Acrescentar um "`by m`" faz com que a lista seja gerada pulando os elementos de m em m. Uma construção ainda mais complexa é possível: podemos utilizar, como em matemática, uma barra vertical `|` para representar *tal que*. Isto nos permite introduzir condições extras que os elementos da lista devam satisfazer. Por exemplo, para gerar os primos entre 1 e 100, basta escrever

```
-> [k for k in 1..100 | prime?(k)]
```

Simples, não? Esta é uma das grandes vantagens de utilizar um sistema de computação algébrica como o AXIOM, em vez de programar diretamente em C. Mesmo utilizando uma biblioteca adequada, não teríamos acesso em C a construções como a anterior, que facilitam enormemente a programação de algoritmos matemáticos, como os que nos interessam neste curso. A bem da verdade, há uma maneira ainda mais fácil de gerar uma lista de primos, bastanto para isto utilizar uma função que tem "prime" como parte do nome. Para discobri-la use `)what op prime`.

5. Laços `while`

Este é, com certeza, o laço que usaremos mais freqüentemente. Por isso, estudaremos vários exemplos de como aplicá-lo.

5.1. **Laço `while`.** Este laço tem a forma geral

`while` *Condição* `repeat`
Instruções

Note que as *Instruções* devem aparecer tabuladas em relação à posição do `while`.

Podemos usá-lo, por exemplo, para calcular o máximo divisor comum entre dois inteiros pelo algoritmo euclidiano. Para isto criamos duas variáveis, digamos R_1 e R_2, às quais serão alocados os valores dos restos anterior e seguinte. Assim, se queremos calcular

$$\text{mdc}(1455, 1239)$$

devemos inicializar as variáveis com estes dois valores:

```
R_1 := 1455
R_2 := 1239
```

Em seguida vem o laço `while`. Observe que, quando R_2 se anular, o valor do máximo divisor comum estará em R_1. Portanto, a condição a ser satisfeita para que o laço continue a ser executado é que $R_2 > 0$. Enquanto isto ocorrer, o algoritmo euclidiano calcula o resto da divisão entre R_1 e R_2; depois aloca o antigo valor de R_2 em R_1 e atribui à variável R_2 o valor do novo resto. Para fazer isto sem perder nenhuma informação, precisaremos de uma variável extra. Juntando tudo isto o laço se torna:

```
while R_2 > 0 repeat
     R:= R_2
     R_2 := rem(R_1,R)
     R_1 := R
```

Só falta mesmo retornar o valor do último resto não nulo, que está alocado à variável R_1. Para isto basta escrever R_1 fora do laço `while`. Resumindo, o programa todo pode ser escrito da seguinte forma:

```
R_1 := 1455
R_2 := 1239
while R_2 > 0 repeat
     R:= R_2
     R_2 := rem(R_1,R)
     R_1 := R
R_1
```

5.2. **Macros.** O problema do programa que acabamos de fazer para o algoritmo euclidiano é que, cada vez que desejamos executá-lo, precisamos abrir o arquivo do programa e alterar a entrada. Para contornar este problema podemos usar `macros`: conjuntos de instruções, aos quais atribuímos um nome. Em geral, um `macro` pode ter ou não parâmetros; mas, da maneira que vamos usá-lo, quase sempre teremos parâmetros. Por exemplo, para criar um macro `mdc` que executa o algoritmo euclidiano, precisaremos de dois parâmetros, que corresponderão aos números dos quais estamos calculando o máximo divisor comum. Por exemplo, nosso programa do algoritmo euclidiano pode ser convertido em um macro, como segue:

```
macro mdc(a,b) ==
     R_1 := a
     R_2 := b
     while R_2 > 0 repeat
          R:= R_2
          R_2 := rem(R_1,R)
          R_1 := R
     R_1
```

Note o uso do `==` que já havia aparecido quando definimos funções. Este macro se chama `mdc`, que é o nome pelo qual será utilizado. Em outras palavras, se o arquivo `mdc.input` contém o macro acima, então depois de carregá-lo usando `)read` podemos calcular o máximo divisor comum de 1455 e 1239 simplesmente escrevendo `mdc(1455,1239)` diretamente no `prompt`.

Exercícios 5.1. *O macro acima funciona corretamente se escolhemos inteiros positivos a e b tais que $a < b$? Explique cuidadosamente sua resposta.*

Nosso conhecimento de macros e do laço `while` é quase suficiente para programarmos o algoritmo de fatoração de Fermat. Falta, apenas, aprendermos a fazer testes, que é o assunto do próximo artigo.

5.3. `if-then-else`. Para fazermos testes, precisamos da construção

```
if condição then
   Instruções
else
   Instruções
```

A parte correspondente ao `else` é opcional e pode ser omitida do teste. Vejamos como esta construção pode ser usada para definir um macro que retorna `primo` se o número n dado for primo e `composto`, se for composto. Para começo de conversa, o AXIOM já dispõe de uma função semelhante, chamada de `primo?`: se n for primo, `primo?(n)` retorna `true`, caso contrário, retorna `false`. Nosso objetivo é alterar apenas a saída da função para que passe a ser `primo` ou `composto`, conforme o caso. Portanto, o macro `ehPrimo` deve:

(1) verificar *se* `primo?(m) = true`;
(2) se for *então* retornar `primo`;
(3) se não for (`else`) retornar `composto`.

Para imprimir `primo` ou `composto` na tela, podemos utilizar o comando `output` já estudado antes. Escrevendo `-> output("primo")` obtemos

```
-> primo
```

como saída. Reunindo todas estas observações, temos o seguinte macro:

```
macro ehPrimo(n) ==
     if prime?(n) = true then
          output("primo")
     else
          output("composto")
```

Com isto podemos converter um de nossos programas anteriores para a forma de um macro.

Exercícios 5.2. *Reescreva o programa para gerar os números N_k, descrito na página 11, na forma de um macro.*

Uma variação mais interessante deste exercício é obtida ao final dos próximos dois exercício.

Exercícios 5.3. *O objetivo deste exercício é obter um macro que conta o número de fatores primos distintos de um dado inteiro n.*

(1) *Investigue o uso correto da função* `factors` *e descubra o que ela retorna.*
(2) *Contando o número de elemento em* `factors(n)` *construa um macro que podemos chamar de* `numeroFatores` *capaz de contar o número de fatores primos* distintos *do inteiro positivo* n.

Exercícios 5.4. *O objetivo deste exercício é construir um macro* `tamanhoNk` *que, ao receber* k*, retorna o número de fatores primos* distintos *do número* N_k *definido na página 11. Proceda como segue:*

(1) *Crie um contador* i*, incializado com* 1*, para determinar quando chegou a hora de parar.*
(2) *Crie outro contador* m*, também incializado com* 1*, cujo valor final será o número total de fatores distintos de* N_k.
(3) *A cada laço, aplique* `numeroFatores` *a* N_k *e some o resultado a* m.
(4) *Repita isto de* i *igual a* 1 *até* k.
(5) *Retorne* m.

No próximo exercício mostramos como construir um macro para o algoritmo de fatoração de Fermat. Lembre-se que, dado o inteiro positivo n, a ser fatorado, a primeira coisa a fazer é calcular a parte inteira de sua raiz quadrada. Contudo `sqrt(n)` tem apenas o efeito de fazer o AXIOM escrever $\sqrt{n}$; isto é, o sistema não calcula uma aproximação do valor da raiz; para forçá-lo a fazer isto determinamos `sqrt(n)` como sendo de tipo `FLOAT` escrevendo `sqrt(n)::FLOAT`. A propósito, você observou que o sistema carrega automaticamente novos *packages* (bibliotecas) à medida que você vai requerendo novas funções? Muita gente prefere que o sistema não mostre a relação dos *packages* que estão sendo carregados; para isto, basta usar `)set message autoload off`.

Outro ponto importante diz respeito ao que deveremos fazer se n tiver uma raiz quadrada inteira. Neste caso, o programa deverá parar de executar neste ponto e retornar o valor da raiz quadrada. Para interromper o programa num dado ponto usamos `break`.

Exercícios 5.5. *Nosso objetivo neste exercício é criar um macro chamado* `fermat` *que, ao receber um inteiro positivo* n *tenta fatorá-lo usando o algoritmo de Fermat. Sugiro que você ponha todos os seus macros em um arquivo de input só, que pode chamar, por exemplo, de* `meusmacros`*. Para criar o macro, proceda da seguinte maneira:*

(1) *Calcule a parte inteira da raiz quadrada de* n*. Para a parte inteira use a função* `wholePart`*. Atribua isto a uma variável* x.
(2) *Faça um teste para ver se* $n - x^2 = 0$*. Se for, interrompa a execução usando* `break` *e retorne* x.

(3) *Se $n - x^2 \neq 0$, então construa um laço* `while` *que deve executar enquanto $\sqrt{n - x^2}$ não for inteiro. Observe, contudo, que isto deve ser testado usando a parte inteira de $\sqrt{n - x^2}$ para evitar problemas com cálculos inexatos.*
(4) *Ao final o programa deve retornar um conjunto com dois fatores de n ou uma mensagem identificando n como primo. Para isto vai ser necessário verificar, ao final, se $x = (n + 1)/2$ ou não, para que seja possível ter certeza se n é ou não primo.*

Exercícios 5.6. *Use* `output` *para reestruturar o programa acima de maneira que retorne, não apenas os fatores de n, mas toda a tabela utilizada para calcular os fatores. A tabela deve ter, o valor de x na primeira coluna, o valor da parte inteira da raiz quadrada de $x^2 - n$ na segunda coluna e o valor de $x^2 - y^2 - n$ na terceira coluna. Note que o algoritmo pára exatamente quando este este último valor se anula.*

5.4. **Crivo de Eratóstenes.** O *Crivo de Eratóstenes* é o programa mais sofisticado que já fizemos até aqui. Para começar, seja $n > 0$ um inteiro e digamos que desejamos achar a lista dos primos positivos menores ou iguais a n. Faremos isto construindo um macro que chamaremos de `Eratostenes`. Procederemos como na seção 3 do capítulo 6 do livro-texto [4].

A primeira coisa a fazer é construir as listas `L` e `Primos`. A primeira conterá $[n/2]$ posições, todas preenchidas com zero. A segunda, será inicializada com vazio. Acrescentaremos os primos à lista `Primos` um a um, à medida que o crivo os for encontrando.

O processo de crivagem propriamente dito contém dois laços. Usaremos laços `while` em ambos os casos. O laço mais externo procura o próximo primo com o qual efetuar a crivagem. Este primo corresponde à próxima casa da lista (digamos que seja a i-ésima) ainda ocupada por 0. Neste caso, o primo correspondente é $2i + 1$ e podemos acrescentá-lo ao final da lista `Primos`. Ao mesmo tempo, determinamos a posição de $(2i + 1)^2$ (o primeiro número múltiplo de $2i + 1$ mas não de um primo menor) na lista `L` como sendo $2(i^2 + i)$.

Com isto podemos executar o laço mais interno, que "risca" os números de $2i + 1$ em $2i + 1$. Lembre-se que, neste contexto, "riscar" é o mesmo que atribuir 1 à casa correspondente da lista. Usando uma nova variável j para controlar o laço mais interno, temos a seguinte construção:

```
j:= 2*(i**2+i)
while j < n+1 repeat
     L.j := 1
     j := j+2*i+1
     i:= i+1
```

Finalmente, não esqueça de escrever `Primos` ao final do macro, e com a tabulação correta, para que o programa retorne a lista de primos desejada.

Referências

[1] D. Augot, *Quelques lignes d'input dans Axiom*, disponível em: `http://www-rocq.inria.fr/codes/Daniel.Augot/axiom_intro.pdf`.
[2] M. Bronstein et al., *Axiom: the 30 year horizon*, disponível em: `http://wiki.axiom-developer.org/Mirrors?go=/public/book2.pdf&it=Axiom+Book`.
[3] Q. Carpent e C. Conil, *Guide d'utilisation pratique d'Axiom*, Université de technologie Belfort-Montbéliard, disponível em: `http://la.riverotte.free.fr/axiom`.
[4] S. C. Coutinho, *Números inteiros e criptografia RSA*, Série de Computação e Matemática nº 2, IMPA e SBM, segunda edição (revisada e ampliada), (2000).
[5] T. Daly, *Axiom: the 30 year horizon, volume 1: tutorial*, Lulu Press (2005).

Departamento de Ciência da Computação, Instituto de Matemática, Universidade Federal do Rio de Janeiro, P.O. Box 68530, 21945-970 Rio de Janeiro, RJ, Brazil.

E-mail address: `collier@impa.br`