Introdução à Programação
uma Abordagem Funcional

Alberto Castro, Crediné Menezes, Cláudia Boeres, Christina Rauber, Thais Castro

Departamento de Ciência da Computação – UFAM
Departamento de Informática – UFES
2005-2016

1. Conceitos Básicos

1.1 INTRODUÇÃO

Neste curso o leitor estará se envolvendo com a aprendizagem de conceitos e métodos básicos para a construção de programas de computador. A abordagem que daremos está voltada para o envolvimento do aprendiz com a solução de problemas ao invés da atitude passiva de ver o que os outros fizeram. Uma questão central que permeia o curso é a de que construir programas é uma tarefa de engenharia, e que, portanto produzirá artefatos com os quais o ser humano terá de conviver. Artefatos estes que devem satisfazer requisitos de qualidade e serem, portanto, passíveis de constatação.

Optamos por desenvolver o curso orientado à descrição de funções, um formalismo bastante conhecido por todos os que chegam a este curso. Esperamos, com isso, atenuar algumas dificuldades típicas do ensino introdutório de programação. Nas seções seguintes apresentamos alguns conceitos básicos que nos parecem importantes ter em mente antes de iniciarmos um curso de programação.

1.2. COMPUTADORES

Denominamos de computador uma máquina de processar dados, numéricos ou simbólicos, que funciona através da execução de programas. Ao contrário das inúmeras máquinas que conhecemos, tais como: de lavar roupa, liquidificador, enceradeira, aspirador de pó, e tantas outras, que realizam uma única função, o computador é uma máquina multiuso. Podemos usá-lo como uma máquina de escrever sofisticada, como uma máquina de fax, como uma prancheta de desenho sofisticada, como um fichário eletrônico, como uma planilha de cálculos e de tantas outras formas. É exatamente como o nosso conhecido videogame: para mudar de jogo basta trocar o cartucho. No videogame, cada novo jogo é determinado por um novo programa.

Em linhas gerais podemos entender um computador como uma máquina capaz de:

a) interpretar dados que lhe são fornecidos, produzindo resultados em forma de novos dados ou comportamentos, usando para isso conceitos que lhe foram antecipadamente informados e,
b) aceitar a descrição de novos conceitos e considerá-los na interpretação de novas situações.

Alguns exemplos de uso de um computador:

Ex 1: Descrever para uma máquina a relação métrica que existe entre os lados de um triângulo retângulo. De posse desse conhecimento, a máquina

poderia, por exemplo, determinar o valor de um dos lados quando conhecido o valor dos outros dois.

Ex 2: Informar a uma máquina as regras de conjugação de verbos. Com este conhecimento a máquina pode determinar a forma correta para um determinado tempo e pessoa de um verbo específico.

Ex 3: Tradução de textos;

Ex 4: Classificação de textos quanto à sua natureza: romance, poesia, documentário, entrevista, artigo científico;

Ex 5: Manipulação de expressões algébricas, resolução de integral indefinida, etc;

Ex 6: Programação automática: dada uma certa especificação, gerar um programa eficiente;

Ex 7: Monitoramento de pacientes em um Centro de Tratamento Intensivo;

Ex 8: Identificação de tumores no cérebro a partir da comparação de imagens com padrões conhecidos de anormalidade;

Ex 9: Roteamento inteligente de mensagens.

1.3. PROGRAMAÇÃO

À tarefa de identificar o conhecimento necessário para a descrição de um conceito, organizá-lo e codificá-lo de modo a ser entendido pela máquina damos o nome de *programação de computadores*. Ao conhecimento codificado, produto final da tarefa de programação dá-se o nome de *programa*.

A programação de computadores é uma atividade que compreende várias outras atividades, tais como: entendimento do problema a ser resolvido, planejamento de uma solução, formalização da solução usando uma linguagem de programação, verificação da conformidade da solução obtida com o problema proposto.

1.4. LINGUAGEM DE PROGRAMAÇÃO

A descrição de conhecimento para um agente racional qualquer (seja uma máquina ou um humano) subentende a existência de padrões segundo os quais o agente possa interpretar o conhecimento informado. A esses padrões, quando rigorosamente elaborados, damos o nome de *formalismo*. Um formalismo é composto de dois aspectos: a sintaxe e a semântica. A sintaxe permite ao agente reconhecer quando uma "sequência de símbolos" que lhe é fornecida está de acordo com as regras de escrita e, portanto representa um programa. A semântica permite que o agente atribua um significado ao conhecimento descrito pela

"sequência de símbolos". Por exemplo, quando um agente humano (com determinado grau de escolaridade) encontra a seguinte seqüência de símbolos {3, 4} U {5, 9, 15}, ele por certo reconhecerá como uma expressão algébrica escrita corretamente e, se lembrar dos fundamentos da teoria dos conjuntos, associará esta cadeia como a descrição de um conjunto composto pela união dos elementos de dois conjuntos menores.

Eis aqui algumas observações importantes sobre a necessidade de linguagens de programação:

- Ainda não é possível usar linguagem natural para ensinar o computador a realizar uma determinada tarefa. A linguagem natural, tão simples para os humanos, possui ambigüidades e redundâncias que a inviabilizam como veículo de comunicação com os computadores.
- A linguagem nativa dos computadores é muito difícil de ser usada, pois requer do programador a preocupação com muitos detalhes específicos da máquina, tirando pois atenção do problema.
- Para facilitar a tarefa de programação foram inventadas as linguagens de programação. Estas linguagens têm por objetivo se colocarem mais próximo do linguajar dos problemas do que do computador em si. Para que o programa que escrevemos possa ser “entendido” pelo computador, existem programas especiais que os traduzem (compiladores) ou os que interpretam (interpretadores) para a linguagem do computador.
- Podemos fazer um paralelo com quando queremos nos comunicar com uma pessoa de língua estrangeira. Podemos escrever uma carta em nossa língua e pedir a alguém que a traduza para a língua de nosso destinatário ou se quisermos conversar pessoalmente, podemos usar um intérprete.

1.5. PROPRIEDADES DE UM PROGRAMA

Fazemos programas com a intenção de dotar uma máquina da capacidade de resolver problemas. Neste sentido, um programa é um produto bem definido, que para ser usado precisa que sejam garantidas algumas propriedades. Aqui fazemos referências a duas delas: a correção e o desempenho. A correção pode ser entendida como a propriedade que assegura que o programa descreve corretamente o conhecimento que tínhamos intenção de descrever. O desempenho trata da propriedade que assegura que o programa usará de forma apropriada o tempo e os recursos da máquina considerada. Cabe aqui alertar aos principiantes que a tarefa de garantir que um programa foi desenvolvido corretamente é tão complexa quanto à própria construção do programa em si. Garantir que um programa funciona corretamente é condição imprescindível para o seu uso e, portanto estaremos dando maior ênfase a esta propriedade.

1.6. PARADIGMAS DE LINGUAGENS DE PROGRAMAÇÃO

As regras que permitem a associação de significados às “sequências de símbolos” obedecem a certos princípios. Existem várias manifestações destes princípios e a cada uma delas denominamos de paradigma.

Um paradigma pode ser entendido informalmente como uma forma específica de se "pensar" sobre programação. Existem três grandes grupos de paradigmas para programação: o procedimental, o funcional e o lógico. Os dois últimos são frequentemente referidos como sendo sub-paradigmas de um outro mais geral, o paradigma declarativo. O paradigma procedimental subentende a organização do conhecimento como uma sequência de tarefas para uma máquina específica. O lógico requer o conhecimento de um formalismo matemático denominado de lógica matemática. O paradigma funcional baseia-se no uso dos princípios das funções matemáticas. De uma forma geral, os paradigmas declarativos enfatizam o aspecto correção e o procedimental os aspectos de desempenho. Vejam que falamos em “enfatizam”, o que quer dizer que apresentam facilidades para descrição e verificação da propriedade considerada. Entretanto, em qualquer caso, o programador deverá sempre garantir que os dois aspectos (correção e desempenho) sejam atendidos.

1.7. PROGRAMAÇÃO FUNCIONAL

Para os fins que aqui nos interessam neste primeiro momento, podemos entender o computador, de uma forma simplificada, como uma máquina capaz de:

a) avaliar expressões escritas segundo regras sintáticas bem definidas, como a das expressões aritméticas que tão bem conhecemos (ex. 3 + 5 - 8) obedecendo à semântica das funções primitivas das quais ela é dotada (por exemplo: as funções aritméticas básicas como somar, subtrair, multiplicar e dividir);

b) aceitar a definição de novas funções e posteriormente considerá-las na avaliação de expressões submetidas à sua avaliação.

Por enquanto, denominaremos o computador de máquina funcional. A seguir apresentamos um exemplo de interação com a nossa Máquina Funcional.

usuário: >>> 3 + 5 / 2

sistema: 5.5

usuário: >>> def f (x,y) : return (x + y) / 2

sistema: ...

usuário: >>> f(3,5) + f(10,40)

sistema: 29.0

Na primeira interação podemos observar que o usuário descreveu uma expressão aritmética e que o sistema avaliou e informou o resultado. Na segunda

interação o usuário descreve, através de uma equação, uma nova função, que ele denominou de **f** e que o sistema acatou a nova definição. Na terceira interação o usuário solicita a avaliação de uma nova expressão aritmética usando o conceito recentemente definido e que o sistema faz a avaliação usando corretamente o novo conceito.

1.8. EXPRESSÕES ARITMÉTICAS

A nossa máquina funcional hipotética entende a sintaxe das expressões aritméticas, com as quais todo aluno universitário já é bem familiarizado e é capaz de avaliá-las usando essas mesmas que regras que já conhecemos.

Sintaxe - Todo operador aritmético pode ser entendido, e aqui o será, como uma função que possui dois parâmetros. A notação usual para as operações aritmética é a infixada, ou seja, símbolo funcional colocado entre os dois operandos. Nada impede de pensarmos nele escritos na forma prefixada, que é a notação usual para funções com número de parâmetros diferente de 2. Por exemplo, podemos escrever "***+ 3 2***" para descrever a soma do número 3 com o número 2. As funções definidas pelo programador devem ser escritas de forma prefixada, como no exemplo de interação acima apresentado. Combinando essas duas formas, infixada e prefixada, podemos escrever expressões bastante sofisticadas.

Avaliação - As expressões aritméticas, como sabemos, são avaliadas de acordo com regras de avaliação bem definidas, efetuando as operações de acordo com suas prioridades. Por exemplo, na expressão "***3 + 5 / 2***" o primeiro operador a ser avaliado será o de divisão (/) e posteriormente o de adição. Se desejarmos mudar essa ordem, podemos usar parênteses em qualquer quantidade, desde que balanceados e em posições apropriadas. Por exemplo, na expressão "***(3 + 5) / 2***", a utilização de parênteses determina que a sub-expressão 3 + 5 terá prioridade na avaliação.

1.9. FUNÇÕES

Podemos entender o conceito de funções como uma associação entre elementos de dois conjuntos A e B de tal forma que para cada elemento de A existe apenas um elemento de B associado. O conjunto A é conhecido como o domínio da função, ou ainda como o conjunto de entrada e o conjunto B é o contra-domínio ou conjunto de saída. Para ser mais preciso, podemos afirmar que uma função *f* , que associa os elementos de um conjunto A aos elementos de um conjunto B, é um conjunto de pares ordenados onde o primeiro elemento do par pertence a A o segundo a B. Exemplos:

a) Seja a função T que associa as vogais do alfabeto com os cinco primeiros inteiros positivos.

   T = {(a,1), (e,2), (i,3), (o,4), (u,5)}

b) Seja a função Q, que associa a cada número natural o seu quadrado.

Q = {(0,0), (1,1), (2,4), (3,9), (4,16), ...}

Podemos observar que a função T é um conjunto finito e que a função Q é um conjunto infinito.

1.10. DESCRIÇÕES FUNCIONAIS

Podemos descrever um conjunto, de duas formas: *extensional*, onde explicitamos todos os elementos que são membros do conjunto, como no caso do conjunto T apresentado anteriormente; ou na forma *intencional*, onde descrevemos um critério de pertinência dos membros do conjunto. Por exemplo, o conjunto Q acima apresentado poderia ser reescrito da seguinte forma:

Q = {(x, y) | x é natural e y = x.x} que pode ser lido da seguinte maneira:

Q é o conjunto dos pares ordenados (x, y) tal que x é um número natural e y é o produto de x por x.

Quando descrevemos uma função para fins computacionais, estamos interessados em explicitar como determinar o segundo elemento do par ordenado, conhecido o primeiro elemento do par. Em outras palavras, como determinar y conhecendo-se o valor de x. Normalmente dizemos que queremos determinar y em função de x. Nesta forma de descrição, omitimos a variável y e explicitamos o primeiro elemento que é denominado então de parâmetro da função. No caso acima teríamos então:

Q *x* = *x* . *x* que utilizando a linguagem Python seria definida como

def Q(x) : return x*x

1.11. Por que começar por programação funcional?

Tendo em vista a prática vigente de começar o ensino de programação utilizando o paradigma procedimental, apresentamos a seguir alguns elementos que baseiam nossa opção de começar o ensino de programação usando o paradigma funcional.

1) O aluno de graduação em Computação tem de 4 a 5 anos para aprender todos os detalhes da área de computação, portanto não se justifica que tudo tenha que ser absorvido no primeiro semestre. O curso introdutório é apenas o primeiro passo e não visa formar completamente um programador. Este é o momento de apresentar-lhe os fundamentos e, além disso, permitir que ele vislumbre a variedade de problemas;

2) O paradigma procedimental requer que o aluno tenha um bom entendimento dos princípios de funcionamento de um computador real, pois eles se baseiam, como as máquinas reais, no conceito de mudança de estados (máquina de Von Neumann).

3) O paradigma lógico, outro forte candidato, requer o conhecimento de lógica matemática que o aluno ainda não domina adequadamente;

4) O paradigma funcional é baseado num conhecimento que o aluno já está familiarizado desde o ensino médio (funções, mapeamento entre domínios) o qual é ainda explorado em outras disciplinas do ciclo básico, o que nos permite concentrar nossa atenção na elaboração de soluções e na descrição formal destas;

5) O elevado poder de expressão das linguagens funcionais permite que a abrangência do uso do computador seja percebida mais rapidamente. Em outras palavras, podemos resolver problemas mais complexos já no primeiro curso;

6) O poder computacional do paradigma funcional é idêntico ao dos outros paradigmas. Apesar disso, ele ainda não é usado nas empresas, por vários aspectos. Dentre os quais podemos citar:

   i) No passado, programas escritos em linguagens funcionais eram executados muito lentamente. Apesar disso não ser mais verdadeiro, ficou a fama;

   ii) A cultura de linguagens procedimentais possui muitos adeptos no mundo inteiro, o que, inevitavelmente, cria uma barreira à introdução de um novo paradigma. Afinal, temos medo do desconhecido e trememos quando temos que nos livrar de algo que já sabemos;

   iii) Há uma crença que linguagens funcionais são difíceis de aprender e só servem para construir programas de inteligência artificial.

7) A ineficiência das linguagens funcionais em comparação às procedimentais tem se reduzido através de alguns mecanismos tais como: *lazy evaluation*, grafo de redução, combinadores.

8) Para fazer um programa que “funciona” (faz alguma coisa, não necessariamente o que desejamos) é mais fácil fazê-lo no paradigma procedimental. Para fazer um programa que funciona “corretamente” para resolver um determinado problema é mais fácil no paradigma funcional, pois esse paradigma descreve “o que fazer” e não “como fazer”.

9) As linguagens funcionais são geralmente utilizadas para processamento simbólico, ou seja, solução de problemas não numéricos. (Exemplo: integração simbólica $\times$ integração numérica). Atualmente constata-se um crescimento acentuado do uso deste tipo de processamento.

10) A crescente difusão do uso do computador nas mais diversas áreas do conhecimento gera um crescimento na demanda de produção de programas cada vez mais complexos. Os defensores da programação funcional acreditam que o uso deste paradigma seja

uma boa resposta a este problema, visto que com linguagens funcionais podemos nos concentrar mais na solução do problema do que nos detalhes de um computador específico, o que aumenta a produtividade.

11) Se mais nada justificar o aprendizado de uma linguagem funcional como primeira linguagem, resta a explicação didática. Estamos dando um primeiro passo no entendimento de programação como um todo. É, portanto, importante que este passo seja simplificado através do apoio em uma “máquina” já conhecida, como é o caso das funções.

12) Ainda um outro argumento: mesmo que tenhamos que usar posteriormente uma outra linguagem para ter uma implementação eficiente, podemos usar o paradigma funcional para formalizar a solução. Sendo assim, a versão em linguagem funcional poderia servir como uma especificação do programa.

Exercícios:

1. Conceitue programação de computadores.

2. Quais os principais paradigmas de programação e o que os diferenciam.

3. Faça uma descrição intencional da função: F = {1,3,5,7,...}.

2. A Linguagem de Programação Python

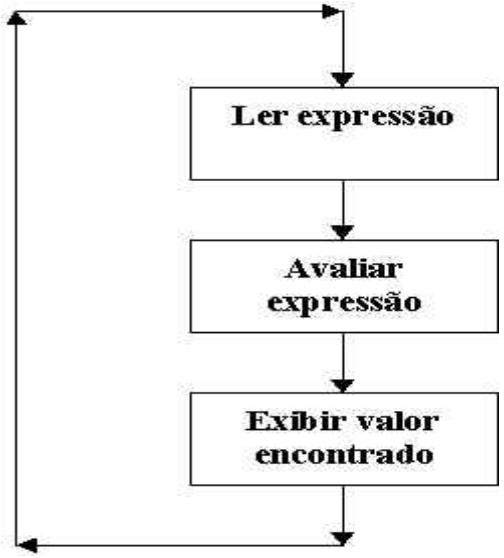
2.1. INTRODUÇÃO: Neste curso usaremos uma implementação da linguagem de programação Python. Essa linguagem, por apresentar uma sintaxe simples e elegante e um conjunto de recursos bastante expressivos, tem sido usada com bons resultados também para a aprendizagem de fundamentos de programação.

Podemos usar o Python como uma calculadora qualquer, à qual submetemos expressões que ela avalia e nos informa o valor resultante. Vejamos por exemplo a interação a seguir.

```
>>> 3 + 5 * 2
    13
>>> (3 + 5) * 2
    16
>>>
```

Nas expressões acima é importante destacar o uso do símbolo * (asterisco) que é empregado para representar a multiplicação. Além desse, outros símbolos usuais serão substituídos, como veremos logo mais. As operações aritméticas são, como sabemos, funções. Entretanto, a notação utilizada em Python é a usual, ou seja, infixada (o símbolo da operação fica entre os operandos). Uma outra forma que pode ser usada para escrever expressões é usando o operador de forma prefixada. Por exemplo, ***abs(-15)***, indica o valor absoluto do número 15.

Uma sequência de símbolos de maior (>>>) é usado pelo sistema para indicar que está preparado para avaliar uma nova expressão. Após avaliar uma expressão o Python informa o resultado na linha seguinte e em seguida exibe uma nova interrogação, se disponibilizando para uma nova avaliação. Podemos explicar o seu funcionamento conforme o esquema da figura abaixo.



2.2. DESCRIÇÃO DE FUNÇÕES: A forma de descrever funções é similar ao que nos referimos anteriormente, ou seja, através de uma equação, onde no lado esquerdo da igualdade damos um nome à função e relacionamos os parâmetros

considerados na sua definição. No lado direito escrevemos uma expressão utilizando outras funções primitivas ou não. Isto nos leva portanto a saber que a linguagem possui funções primitivas que já a acompanham e que portanto prescindem de definição.

Por exemplo, para definir a função que determina o espaço percorrido por um móvel em movimento retilíneo uniforme, conhecidos a sua velocidade e o tempo decorrido, podemos escrever:

`def` espaço	`( v , t )`	`:`	`return v*t`
nome da função	parâmetros		expressão aritmética que define a relação que há entre os parâmetros
interface da função			corpo da definição

Para “alimentar” o Python com novas definições devemos criar um arquivo em disco no qual editaremos as definições desejadas. Cada arquivo pode ter uma ou mais definições. Normalmente agrupamos em um mesmo arquivo as definições relacionadas com a solução de um problema específico ou definições de propósito geral.

A alimentação de novas definições é indicada ao sistema através de um comando que é usado no lugar de uma expressão. Nesse instante utilizaremos o comando ***from <arq> import ****, indicando que do arquivo cujo nome é indicado em <arq>, todas as definições serão adicionadas à sessão corrente.

Por exemplo, para alimentar as definições contidas num arquivo de nome ***pf001.py***, podemos escrever:

```
>>> from pf001.py import *
```

A partir deste momento, todas as definições contidas no arquivo informado estarão disponíveis para uso. Podemos entender isso como fazer uma extensão da “máquina” Python.

2.3. UM EXEMPLO: Considere que queremos descrever uma função para determinar as raízes de uma equação do segundo grau.

Sabemos que pela nossa clássica fórmula as raízes são descritas genericamente por:

$$x = \frac{-b \pm \sqrt{b^2 - 4 \times a \times c}}{2 \times a}$$

A solução, como sabemos, é formada por um par de valores. Por enquanto vamos descrever este fato por duas funções, uma para a primeira raiz e outra para a segunda.

```
def quad(x): return x * x
def eq2g1(a,b,c):
    return ((-b) + sqrt (quad(b) - 4.0*a*c )) / (2.0*a)
```

Vamos discutir alguns detalhes desta codificação:

o termo ***-b*** precisa ser codificado entre parêntesis pois números negativos são obtidos por um operação unária que produz um número negativo a partir de um positivo;

o símbolo da raiz quadrada foi substituído pela função ***sqrt***;

o numerador da fração precisa ser delitimado pelo uso de parêntesis;

o denominador também precisa ser delimitado por parêntesis;

o símbolo de multiplicação usual foi trocado pelo * (asterisco).

Podemos agora descrever a outra raiz de forma análoga:

```
def eq2g2(a,b,c):
    return ((-b) – sqrt(quad(b) - 4.0*a*c))/(2.0*a)
```

Visto que as duas possuem partes comuns, poderíamos ter escrito abstrações auxiliares e produzir um conjunto de definições, tal como:

```
def           quad(x) : return x*x
def  raizdelta(a,b,c) : return sqrt ( quad(b) - 4.0*a*c )
def          dobro(x) : return 2.0*x
def      eq2g1(a,b,c) : return ((-b) + raizdelta(a,b,c)) / dobro(a)
def      eq2g2(a,b,c) : return ((-b) - raizdelta(a,b,c)) / dobro(a)
```

Vejamos como ficaria uma interação com o Python, a partir de um arquivo de definições denominado eq2g.py:

```
>>> from eq2g import *
>>> eq2g1(2,5.0,2)
-0.5
>>> eq2g2(2,5.0,2)
-2.0
>>> eq2g1(3.0,4.0,5.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/alberto/eq2g.py", line 6, in eq2g1
    def   eq2g1(a,b,c)   : return  ((-b) + raizdelta(a,b,c))
/ dobro(a)
  File "/Users/alberto/eq2g.py", line 4, in raizdelta
    def   raizdelta(a,b,c)    : return  sqrt ( quad(b) -
4.0*a*c )
ValueError: math domain error
>>>
Program error: {sqrt (-44.0)}
```

Podemos observar que houve um problema com a avaliação da expressão ***eq2g1(3.0,4.0,5.0)***

Este é um erro semântico, provocado pela tentativa de extrair a raiz quadrada de um número negativo. A função que definimos portanto é uma função

parcial, ou seja, ela não está definida para todo o domínio dos reais. Neste caso o sistema apenas acusa o problema ocorrido.

2.4. DEFINIÇÕES LOCAIS: As definições que discutimos anteriormente são globais, no sentido de que estão acessíveis ao uso direto do usuário e também disponíveis para uso em qualquer outra definição. Por exemplo, se temos o ***script*:**

```
def quad(x):   x * x
def hipo(x,y): sqrt(quad(x) + quad(y))
```

Podemos utilizar a definição ***quad*** tanto externamente pelo usuário quanto internamente pela definição ***hipo***.

Se por outro lado desejamos construir sub-definições para uso em uma definição específica, podemos defini-las internamente, ou seja, alinhadas no lado direito da definição. A maneira de introduzir definições locais é utilizando a cláusula ***where.*** Vamos modificar o script acima para que ***quad*** só possa ser usado no interior de ***hipo***.

```
def hipo(x,y):
    def quad(x):
        return (x*x)
    return sqrt(quad(x)+quad(y))
```

As definições internas também não precisam ser paramétricas. No exemplo que estamos considerando podemos escrever:

```
def hipo(x,y) :
    def k1() : return x*x
    def k2() : return y*y
    return sqrt(k1()+k2())
```

Note que apesar de ***x*** e ***y*** não serem parâmetros de ***k1*** e ***k2*** eles foram utilizados em suas definições. Isto é possível porque ***x*** e ***y*** têm validade em todo o lado direito da definição.

Temos que considerar ainda que nada impede que em uma definição local tenhamos uma outra definição local e dentro desta outra, e assim sucessivamente.

2.5. MODELO DE AVALIAÇÃO DE EXPRESSÕES: Quando o Python toma uma expressão contendo apenas constantes e operações primitivas, ele apenas efetua as operações obedecendo à prioridade dos operadores e aquelas determinadas pelo uso de parêntesis. Por exemplo, para avaliar a expressão 3 + 5 / 2, primeiro é realizada a divisão 5/2, resultando em 2.5, o qual é adicionado ao 3, finalmente obtendo 5.5. Podemos entender isso como uma sequência de reduções de uma expressão à outra mais simples, até que se atinja um termo irredutível. Veja os exemplos a seguir:

3 + 5 / 2 ==> 3 + 2.5 ==> 5.5
(3 + 5) / 2 ==> 8 / 2 ==> 4
3 + 5 + 2 ==> 8 + 2 ==> 10

Quando as expressões utilizam definições, o processo é análogo. Cada referência à uma definição é substituída por seu lado direito, até que se atinja uma expressão básica, e continua como no caso anterior. Vejamos os exemplos abaixo, considerando a primeira definição de ***hipo*** e de ***quad*** apresentada anteriormente:

ordem	expressão	redução aplicada
1	hipo(3,5) + hipo(4,4)	expressão inicial
2	sqrt(quad(3) + quad(5)) + hipo(4,4)	def de hipo
3	sqrt(3*3 + quad(5)) + hipo(4,4)	def de quad
4	sqrt(3*3 + 5*5) + hipo(4,4)	def de quad
5	sqrt(3*3 + 5*5) + sqrt(quad(4) + quad(4))	def de hipo
6	sqrt(3*3 + 5*5) + sqrt(4*4 + quad (4))	def de quad
7	sqrt(3*3 + 5*5) + sqrt(4*4 + 4*4)	def de quad
8	sqrt(9 + 5*5) + sqrt(4*4 + 4*4)	*
9	sqrt(9 + 25) + sqrt(4*4 + 4*4)	*
10	sqrt(34) + sqrt(4*4 + 4*4)	+
11	5.83095 + sqrt(4*4 + 4*4)	sqrt
12	5.83095 + sqrt(16 + 4*4)	*
13	5.83095 + sqrt(16 + 16)	*
14	5.83095 + sqrt(32)	+
15	5.83095 + 5.65685	sqrt
16	11.4878	+

Para uma realização específica da linguagem, isso não precisa acontecer exatamente assim. Entretanto este modelo é suficiente para nossos interesses. O número de reduções necessárias para chegar à forma irredutível de uma expressão, também denominada de forma canônica ou ainda forma normal, pode ser usado como critério para discutir o desempenho da mesma.

3. A Arte de Resolver Problemas

3.1. INTRODUÇÃO: O grande objetivo deste curso é criar oportunidades para que o estudante desenvolva suas habilidades como resolvedor de problemas. Mais especificamente estamos interessados na resolução de problemas usando o computador, entretanto, temos certeza, que as ideias são gerais o suficiente para ajudar a resolver qualquer problema.

Embora muitas pessoas já tenham discutido sobre este assunto ao longo da história, nosso principal referencial será o professor George Polya, que escreveu um livro sobre este assunto, com o mesmo nome deste capítulo, voltado para a resolução de problemas de matemática do ensino fundamental. Todos os alunos deste curso terão grande proveito se lerem este livro.

As ideias ali apresentadas com certeza se aplicam com muita propriedade à resolução de problemas com o computador. Na medida do possível estaremos apresentando aqui algumas adaptações que nos parecem apropriadas neste primeiro contato com a resolução de problemas usando o computador.

3.2. DICAS INICIAIS: apresenta-se a seguir algumas orientações:

3.2.1. Só se aprende a resolver problemas através da experiência. Logo o aluno que está interessado em aprender deve trabalhar, buscando resolver por si mesmo, os problemas propostos antes de tentar o auxílio do professor ou de outro colega;

3.2.2. A ajuda do professor não deve vir através da apresentação pura e simples de uma solução. A ajuda deve vir através do fornecimento de pistas que ajudem o aluno a descobrir a solução por si mesmo;

3.2.3. É muito importante não se conformar com uma única solução. Quanto mais soluções encontrar, mais hábil o estudante ficará, e além disso, poderá comparar as várias alternativas e escolher a que lhe parecer mais apropriada. A escolha deverá sempre ser baseada em critérios objetivos.

3.2.4. Na busca pela solução de um problema, nossa ferramenta principal é o questionamento. Devemos buscar formular questões que nos ajudem a entender o problema e a elaborar a solução.

3.2.5. Aprenda desde cedo a buscar um aprimoramento da sua técnica para resolver problemas, crie uma sistematização, evite chegar na frente de um problema como se nunca tivesse resolvido qualquer outro problema. Construa um processo individual e vá aperfeiçoando-o à cada vez que resolver um novo problema.

3.3 COMO RESOLVER UM PROBLEMA: O professor Polya descreve a resolução de um problema como um processo complexo que se divide em quatro etapas, as quais apresentamos aqui, com alguma adaptação. Segundo nos recomenda Polya, em qualquer destas etapas devemos ter em mente três questões sobre o andamento do nosso trabalho: Por onde começar esta etapa? O que posso fazer com os elementos que disponho? Qual a vantagem de proceder da forma escolhida?

Fase 1 - Compreensão do Problema

É impossível resolver um problema sobre o qual não tenhamos um entendimento adequado. Portanto, antes de correr atrás de uma solução, concentre-se um pouco em identificar os elementos do problema. Faça perguntas tais como: Quais são os dados de entrada? O que desejamos produzir como resultado? Qual a relação que existe entre os dados de entrada e o resultado a ser produzido? Quais são as propriedades importantes que os dados de entrada possuem?

Fase 2 - Planejamento

Nesta fase devemos nos envolver com a busca de uma solução para o problema proposto. Pode ser que já o tenhamos resolvido antes, para dados ligeiramente modificados. Se não é o caso, pode ser que já tenhamos resolvido um problema parecido. Qual a relação que existe entre este problema que temos e um outro problema já conhecido? Será que podemos quebrar o problema em problemas menores? Será que generalizando o problema não chegaremos a um outro já conhecido? Conhecemos um problema parecido, embora mais simples, o qual quando generalizado se aproxima do que temos?

Fase 3 - Desenvolvimento

Escolhida uma estratégia para atacar o problema, devemos então caminhar para a construção da solução. Nesta fase, devemos considerar os elementos da linguagem de programação que iremos usar, respeitando os elementos disponibilizados pela linguagem, tais como tipos, formas de definição, possibilidades de generalização e uso de elementos anteriormente definidos. A seguir, devemos codificar cada pedaço da solução, e garantir que cada um esteja descrito de forma apropriada. Em outras palavras, não basta construir, temos que ser capazes de verificar se o resultado de nossa construção está correto. Isto pode ser realizado pela identificação de instâncias típicas, da determinação dos valores esperados por estas, da submissão dessa instâncias ao avaliador e finalmente, da comparação dos resultados esperados com os resultados produzidos pela avaliação de nossas definições. Além disso, devemos garantir que a definição principal também está correta.

Em síntese, a fase de desenvolvimento compreende as seguintes subfases:

construção da solução;
planejamento do teste;
teste com o computador de mesa;
codificação da solução;
teste com o uso do computador.

Fase 4 - Avaliação do Processo e seus resultados

O trabalho do resolvedor de problemas não para quando uma solução está pronta. Devemos avaliar a qualidade da solução, o processo que realizamos e questionar as possibilidades de uso posterior da solução obtida e também do próprio método utilizado. Novamente devemos fazer perguntas: Este foi o melhor caminho que poderia ter sido usado? Será que desdobrando a solução não obtenho componentes que poderei usar mais facilmente no futuro? Se esta

solução for generalizada é possível reusá-la mais facilmente em outras situações? Registre tudo, organize-se para a resolução de outros problemas. Anote suas decisões, enriqueça a sua biblioteca de soluções e métodos.

3.4. UM PEQUENO EXEMPLO:

Enunciado: Deseja-se determinar a menor quantidade de cédulas necessárias para pagar uma quantia em Reais.

Etapa 1 – Compreensão

- Quais os dados de entrada?
  - A quantia a ser paga.
- Qual o resultado a ser obtido?
  - A menor quantidade de cédulas.

- Qual a relação que existe entre a entrada e a saída?
  - O somatório dos valores de cada cédula utilizada deve ser igual à quantia a ser paga.
- Existe algum elemento interno, ou seja, uma dado implícito?
  - Sim, os tipos de cédulas existentes. O Real possui as seguintes cédulas: 2, 5, 10, 20, 50 e 100. É importante observar que qualquer cédula é um múltiplo de qualquer uma das menores.

Etapa 2 - Planejamento

Conheço algum problema parecido? Existe um problema mais simples? Podemos entender como um problema mais simples, um que busque determinar quantas cédulas de um dado valor são necessárias para pagar a quantia desejada. Por exemplo, para pagar R$289,00 poderíamos usar 145 cédulas de R$2,00 mas nesse caso ficaria sobrando R$1,00 ou poderíamos usar 5 notas de R$50,00 mas ficariam faltando R$ 37,00. Claro que não queremos que falte nem que sobre e, além disso, desejamos que a quantidade seja mínima. Parece que uma boa estratégia é começar vendo o que dá para pagar com a maior cédula e determinar quando falta pagar. O restante pode ser pago com uma cédula menor, e por aí afora. Explorando a instância do problema já mencionada, vamos fazer uma tabela com estes elementos.

Expressão para determinar a quantidade de cédulas de um determinado valor.	Quantidade de cédulas	Quantia a Pagar
		289,00
289 / 100	2	89,00
89 / 50	1	39,00
39 / 20	1	19,00

19 / 10	1	9,00
9 / 5	1	4,00
4 / 2	2	0,00
TOTAL	**8**	

Etapa 3 - Desenvolvimento

Solução 1 - Considerando a tabela acima descrita, codificando cada linha como uma subexpressão da definição.

```
def ncedulas(q):
return (
 (q // 100) +
 (q % 100) // 50 +
 ((q % 100) % 50) // 20 +
 (((q % 100) % 50) % 20) // 10 +
 ((((q % 100) % 50) % 20) % 10) // 5 +
 (((((q % 100) % 50) % 20) % 10) % 5) // 2 +
 ((((((q % 100) % 50) % 20) % 10) % 5) % 2))
```

Solução 2 - Considerando uma propriedade das cédulas, ou seja, já que uma cédula qualquer é múltiplo das menores, a determinação do resto não precisa considerar as cédulas maiores do que a cédula que estamos considerando em um dado ponto.

Etapa 4 - Avaliação

A solução deixa de explicitar as abstrações referentes à quantidade de cédulas de um determinado valor, assim como o resto correspondente. Podemos questionar, não seria melhor explicitar? Assim poderíamos usá-las de forma independente e além disso, a solução fica mais clara e portanto inteligível.

```
def ncedulas(q) : return
                    n100(q)+n50(q)+n20(q)+n10(q)+n5(q)+n2(q)
    def n100(q) : return q // 100
    def r100(q) : return q % 100
     def n50(q) : return r100(q) // 50
     def r50(q) : return r100(q) % 50
     def n20(q) : return r50(q) // 20
     def r20(q) : return r50(q) % 20
     def n10(q) : return r20(q) // 10
     def r10(q) : return r20(q) % 10
      def n5(q) : return r10(q) // 5
```

```
      def r5(q) : return r10(q) % 5
      def n2(q) : return r5(q) // 2
```

Supondo que não queremos generalizar todas as funções menores ainda poderíamos escrever o programa usando definições locais.

```
def
ncedulas(q): def n100(q)    :    return q // 100
             def r100(q)    :    return q % 100
             def n50(q)     :    return r100(q) // 50
             def r50(q)     :    return r100(q) % 50
             def n20(q)     :    return r50(q) // 20
             def r20(q)     :    return r50(q) % 20
             def n10(q)     :    return r20(q) // 10
             def r10(q)     :    return r20(q) % 10
             def n5(q)      :    return r10(q) // 5
             def r5(q)      :    return r10(q) % 5
             def n2(q)      :    return r5(q) // 2
             return n100(q)+n50(q)+n20(q)+n10(q)+n5(q)+n2(q)
```

Podemos ainda considerar que se houver uma troca no sistema, de modo a incluir uma nova cédula que não seja múltiplo de seus valores menores. Seria fácil mudar o programa para contemplar a mudança nas leis do mundo do problema.

3.5. PROVÉRBIOS: O professor Polya também nos sugere que a lembrança de alguns provérbios pode ajudar o aprendiz (e o resolvedor de problemas) a organizar o seu trabalho. Diz Polya que, apesar dos provérbios não se constituírem em fonte de sabedoria universalmente aplicável, seria uma pena desprezar a descrição pitoresca dos métodos heurísticos que apresentam.

Alguns são de ordem geral, tais como:

- *O fim indica aos meios.*
- *Seus melhores amigos são "O que", "Por que", "Onde", "Quando" e "Como". pergunte O que, pergunte Por que, pergunte Onde, pergunte Quando e pergunte Como - e não pergunte a ninguém quando precisar de conselho.*
- *Não confie em coisa alguma, mas só duvide daquilo que merecer dúvida.*
- *Olhe em torno quando encontrar o primeiro cogumelo ou fizer a primeira descoberta; ambos surgem em grupos.*

A seguir apresentamos uma lista deles, organizados pelas etapas às quais parecem mais relacionados.

Etapa 1 : Compreensão do problema.

- *Quem entende mal, mal responde.*

- *Pense no fim antes de começar.*
- *O tolo olha para o começo, o sábio vê o fim.*
- *O sábio começa pelo fim, o tolo termina no começo.*

Etapa 2 : Planejamento da solução.

- *A perseverança é a mãe da boa sorte.*
- *Não se derruba um carvalho com uma só machadada.*
- *Se no princípio não conseguir, continue tentando.*
- *Experimente todas as chaves do molho.*
- *Veleja-se conforme o vento.*
- *Façamos com pudermos se não pudermos fazer como queremos.*
- *O sábio muda de opinião, o tolo nunca.*
- *Mantenha duas cordas para um arco.*
- *Faça e refaça que o dia é bastante longo.*
- *O objetivo da pescaria não é lançar o anzol mas sim pegar o peixe.*
- *O sábio cria mais oportunidades do que as encontra.*
- *O sábio faz ferramentas daquilo que lhe cai às mãos.*
- *Fique sempre de olho na grande ocasião.*

Etapa 3: Construção da solução.

- *Olhe antes de saltar.*
- *Prove antes de confiar.*
- *Uma demora prudente torna o caminho seguro.*
- *Quem quiser navegar sem risco, não se faça ao mar.*
- *Faça o que puder e espere pelo melhor.*
- *É fácil acreditar naquilo que se deseja.*
- *Degrau a degrau sobe-se a escada.*
- *O que o tolo faz no fim, o sábio faz no princípio.*

Etapa 4: Avaliação da solução.

- *Não pensa bem quem não repensa.*
- *É mais seguro ancorar com dois ferros.*

4. Abstração, Generalização, Instanciação e Modularização

4.1. INTRODUÇÃO: Na busca por resolver um problema podemos usar vários princípios, cada um evidentemente terá uma utilidade para a resolução do problema, ou para garantia de sua correção ou ainda para facilitar os usos posteriores da solução que obtivermos. Apresentamos a seguir alguns deles.

4.2. ABSTRAÇÃO: Quando escrevemos uma expressão e não damos nome a ela, o seu uso fica limitado àquele instante específico. Por exemplo, suponha que desejamos determinar a hipotenusa de um triângulo retângulo com catetos 10 e 4. Como conhecemos o teorema de Pitágoras, podemos usar diretamente nossa máquina funcional para avaliar a seguinte expressão:

```
>>> sqrt ((10.0 * 10.0)+ (4.0 * 4.0))
10.770329614269007
>>>
```

A expressão que codificamos serve apenas para esta vez. Se em algum outro instante precisarmos usá-la outra vez teremos de codificá-la novamente. Para evitar isso, é que damos nomes às nossas expressões, para que depois possamos usá-las repetidamente, apenas referenciando-as pelo seu nome. No caso acima, poderíamos escrever a definição:

```
>>> def hipo104(): return sqrt((10*10)+(4*4))
...
>>>
```

De posse dessa definição nossa máquina poderá avaliar a expressão sempre que dela precisarmos. Você poderia agora estar pensando, porque não trocamos a definição para usar o valor ***10.7703***? Agindo assim a máquina não precisaria avaliar a expressão ***sqrt ((10.0 * 10.0)+ (4.0 * 4.0))*** a cada uso. Por outro lado não ficaria registrada a origem do valor ***10.7703***.

4.3. GENERALIZAÇÃO: Quando nossa definição se aplica a vários valores podemos generalizá-la, além de abstraí-la. Assim, além de usá-la várias vezes para os mesmos valores, podemos também usá-la para valores diferentes. Podemos fazer isso de duas formas.

Na primeira, usamos outras constantes previamente definidas. Por exemplo, no caso acima, poderíamos escrever:

```
>>> b=10
>>> c=4
>>> def hipo(): return sqrt((b*b)+(c*c))
...
>>>
```

Aqui, ***hipo*** se aplica a dois valores quaisquer ***b*** e ***c***, os quais são objetos de outras definições.

Uma forma mais geral é através da parametrização. Esta consiste em indicar no cabeçalho da definição (lado esquerdo da igualdade) quais são os valores objetos da generalização. No caso presente podemos escrever:

```
>>> def hipo(b,c): return sqrt((b*b)+(c*c))
...
>>>
```

Temos então descrito a função paramétrica ***hipo*** cujos parâmetros são ***b*** e ***c***.

4.4. INSTANCIAÇÃO: A parametrização permite que usemos a mesma definição para diferentes instâncias do problema. Por exemplo, suponha que desejamos determinar a hipotenusa de três triângulos retângulos. O primeiro com catetos 10 e 4, o segundo com catetos 35 e 18 e o terceiro com catetos 9 e 12. Podemos instanciar a definição paramétrica para estabelecer a seguinte interação:

```
>>> hipo(10,4)
10.770329614269007
>>> hipo(35,18)
39.35733730830886
>>> hipo(9,12)
15.0
>>>
```

4.5. MODULARIZAÇÃO: Em geral, nossos problemas não serão tão simples e diretos quanto o exemplo acima. Quando nos deparamos com problemas maiores, um bom princípio é:

Divida para facilitar a conquista.

Basicamente este princípio consiste em quebrar o problema inicial em problemas menores, elaborar a solução para cada um dos problemas menores e depois combiná-las para obter a solução do problema inicial. A cada um dos subproblemas encontrados podemos reaplicar o mesmo princípio. Segundo Polya, esta é uma heurística muito importante à qual ele denomina de Decomposição e Combinação. Antes de pensar em codificar a solução em uma linguagem de programação específica, podemos representá-la através de um esquema gráfico denominado de estrutura modular do problema (veja a representação para o problema a seguir).

4.6. MODULARIZANDO: Considere o problema de descrever área da Figura 3 abaixo.

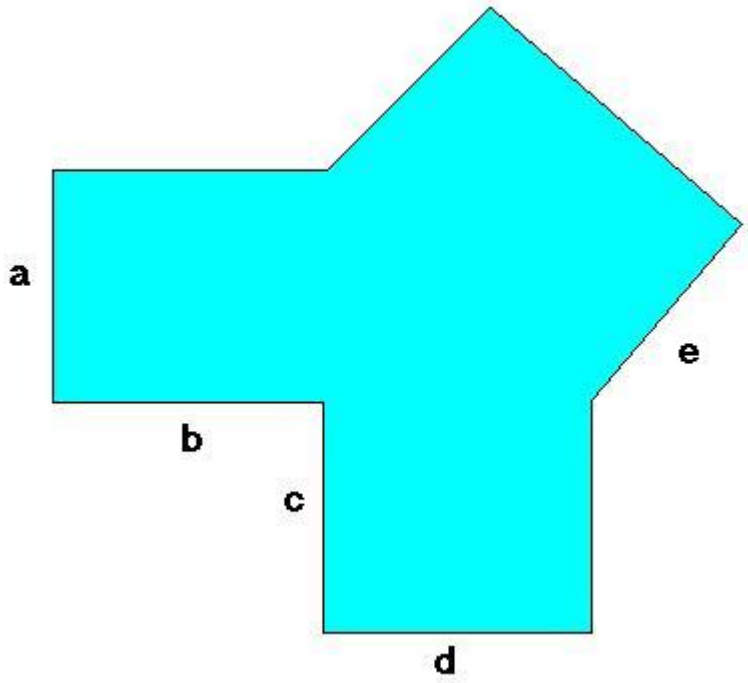


Figura 3 – Cálculo de área

Como podemos concluir por uma inspeção da figura, não existe uma fórmula pronta para calcular sua área. Precisamos dividir a figura em partes para as quais conheçamos uma maneira de calcular a área e que, além disso, conheçamos as dimensões necessárias.

Podemos fazer várias tentativas, como por exemplo, as ilustradas nas figuras Fig.4a, Fig.4b e Fig.4c. Podemos tentar descrever a área das figuras menores em cada uma dela. Em 4b e 4c, parece que precisaremos subdividir novamente, em 4b a casinha em verde pode ser transformada em um retângulo e um triângulo. Em 4c é a casinha azul (meia-água) que pode ser dividida em um retângulo e um triângulo. E a figura Fig. 4a, que podemos dizer?

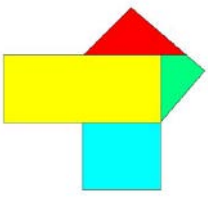


Fig.4a

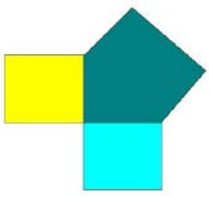


Fig.4b

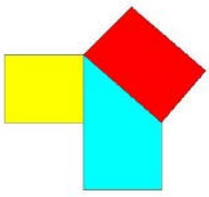


Fig.4c

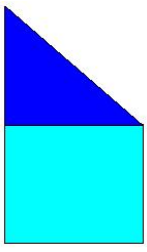


Fig.4d

Vamos partir para a nossa solução a partir da figura Fig.4c. Podemos dizer que a área total pode ser obtida pela soma das áreas amarela, com área vermelha e mais a área azul. A área azul pode ser subdividida em azul-claro e azul-escuro

área total	=	área amarela + área vermelha + área azul
área azul	=	área azul-claro + área azul-escuro

Quando escolhemos as áreas acima citadas, não foi por acaso. A escolha foi baseada na simplicidade do subproblema. Podemos usar este conhecimento para chegar a um outro. Três das áreas que precisamos calcular são de forma retangular. Ou seja, são especializações de um conceito mais geral. A quarta área, também pode ser obtida pela especialização do conceito de triângulo retângulo.

Vamos agora aproximar nossas definições da linguagem de programação.

Portanto podemos escrever:

```
>>> def a_retangulo(x,y): return x*y
...
>>>
```

O triângulo que temos é retângulo e podemos descrever sua área por:

```
>>> def a_t_retangulo(x,y): return (x*y)/2
...
>>>
```

A determinação da área vermelha nos traz uma pequena dificuldade... não nos foi informado qual o comprimento da base do retângulo. E agora? Observando bem a figura podemos concluir que a base do retângulo é igual à hipotenusa do triângulo retângulo de catetos ***a*** e ***d.***

Podemos escrever então:

```
def atotal(a,b,c,d,e) : return a_retangulo(a,b)+a_retangulo(hipo(a,d),e)+a_azul(a,c,d)
def a_azul(x,y,z) : return a_retangulo(y,z)+a_t_retangulo(x,y)
def hipo(x,y) : return sqrt(x*x + y*y)
def a_retangulo(x,y) : return x*y
def a_t_retangulo(x,y) : return (x*y)/2
```

A estrutura da solução de um problema obtida pela modularização pode ser representada por um diagrama denominado de “árvore”. Para o problema acima discutido a árvore é da forma apresentada na Figura 5.

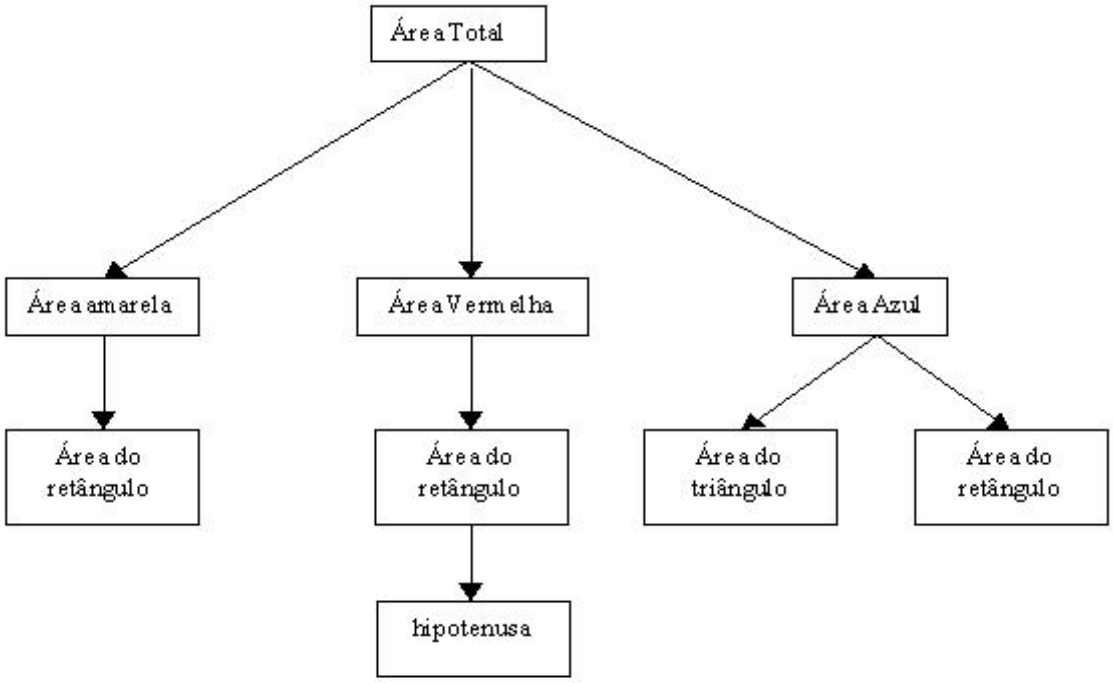


Figura 5 – árvore de solução para cálculo de área

4.7. UM EXEMPLO DETALHADO: Escreva uma descrição funcional para o volume de cada uma das peças apresentadas nas figuras abaixo.

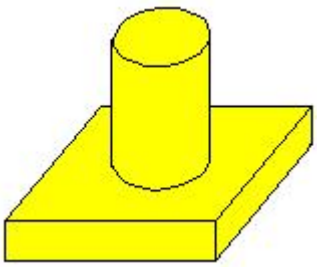


Fig. 6a

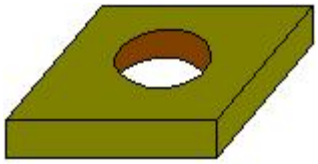


Fig. 6b

Solução 1: Vamos começar pela Fig.6a. Uma rápida análise nos leva a identificar duas partes: Na parte inferior temos um paralelepípedo, sobre o qual se assenta um cilindro. Vamos supor que são maciços. Chamemos de ***a***, ***b*** e ***c*** as dimensões do paralelepípedo, de ***r*** o raio da base do cilindro e de ***h*** a sua altura. O volume total da peça pode ser determinado pela soma do volume das duas peças menores. Chamemos a função de **volfig1**, e tentemos uma definição.

```
>>> def volfig1(a,b,c,r,h): return (a*b*c)+(pi*r*r*h)
...
>>>
```

Vamos então tratar da peça descrita na figura Fig.6b. Agora identificamos uma peça apenas. Um paralelepípedo com um furo no centro. Chamemos de ***a***, ***b*** e ***c*** as dimensões do paralelepípedo, de ***r*** o raio do buraco. Para descrever o volume da peça devemos subtrair do volume do paralelepípedo o volume correspondente ao buraco. Chamemos a função de **volfig2**, e tentemos uma definição.

```
>>> def volfig2(a,b,c,r): return (a*b*c)-(pi*r*r*c)
...
>>>
```

Solução 2: A solução 1, apesar de resolver o problema, deixa de contemplar algumas práticas importantes. No tratamento da primeira peça, apesar de identificadas duas peças menores, estas abstrações não foram descritas separadamente. Ou seja, deixamos de registrar formalmente a existência de duas abstrações e com isso não pudemos modularizar a descrição da função principal. Podemos tentar então um outro caminho:

```
def volcil(r,h)       : return pi*r*r*h
def volpar(a,b,c)     : return a*b*c
def volfig1(a,b,c,r,h) : return volpar(a,b,c)+volcil(r,h)
```

Voltemos então para a segunda peça, e vejamos se podemos identificar similaridades. Aqui só temos uma peça, que como sabemos, guarda uma relação com o paralelepípedo da primeira e só. Como podemos encontrar similaridades? Aprofundando melhor nossa análise podemos lembrar que o furo no paralelepípedo, na verdade corresponde a um cilindro que foi retirado. Assim considerando, podemos então concluir que o volume da figura Fig.6b pode ser obtido pela diferença entre o volume de um paralelepípedo e de um cilindro. Como já temos as definições para volume de cilindro e volume de paralelepípedo, só nos resta escrever a definição final do volume da figura.

```
>>> def volfig2(a,b,c,r,h): return volpar(a,b,c)-volcil(r,h)
...
>>>
```

Analisando a solução, podemos tirar algumas conclusões:

- a solução ficou mais clara,
- a solução propiciou o reaproveitamento de definições.

- se precisarmos usar volume de cilindro e de paralelepípedo isoladamente ou em outras combinações, já as temos disponíveis.

Considerações complementares - assim como decidimos descrever isoladamente o volume de dois sólidos, poderíamos ter feito o mesmo para as figuras. As descrições das áreas de um retângulo e de uma circunferência podem ser dadas isoladamente por:

```
>>> def acir(r): return pi*r*r
...
>>> def aret(a,b): return a*b
...
>>>
```

Podemos reescrever o volume do cilindro e do paralelepípedo da seguinte maneira:

```
>>> def volcil(r,h): return acir(r)*h
...
>>> def volpar(a,b,c): return aret(a,b)*c
...
>>>
```

Solução 3: Se aprofundarmos a análise podemos observar que as duas figuras podem ser abstraídas como uma só! O cilindro que aparece na primeira, como um volume que deve ser acrescentado pode ser entendido como o mesmo cilindro que deve ser subtraído na segunda. Além disso, podemos estender a solução para furos que não vazem a peça, pois a altura do cilindro pode ser qualquer.

```
>>> def volfig(a,b,c,r,h): return volpar(a,b,c)+volcil(r,h)
...
>>>
```

Podemos observar que esta é a mesma definição apresentada anteriormente para a figura Fig.6a. O que muda é no uso. Para calcular o volume de Fig.6a usamos um ***h*** positivo e para Fig.6b um ***h*** negativo.

`>>> volfig(6,8,2,2,5)`
`158.83185307179588`
`>>>`
	-- determinar o volume de uma instância da figura Fig.6a, com um cilindro de raio 2 e altura 5.

`>>> volfig(6,8,2,2,-2)`
`70.86725877128166`
`>>>`
	-- determinar o volume de uma instância da figura Fig.6b, vasada por um furo de raio 2.
	-- neste caso fornecemos um valor negativo para a altura do cilindro com o

	mesmo valor da altura do paralelepípedo

`>>> volfig(6,8,2,2,-1)`
`83.43362938564083`
`>>>`
	-- determinar o volume de uma instância da figura Fig.6b, que tem um furo de raio 2 e profundidade 1.

Façamos agora um pequeno questionamento. Qual é o significado da expressão:

`volfig(6,8,2,2,-5)`

5. Tipos de Dados Numéricos

5.1. INTRODUÇÃO: Uma expressão é composta de valores e símbolos funcionais. Os símbolos funcionais não se aplicam a quaisquer valores e sim apenas àqueles para os quais estão definidos. Relembrando, uma função possui um domínio e um contra-domínio. Aqui entendemos Tipo de Dado como um conjunto de valores definido como domínio ou contra-domínio.

Um tipo de dado envolve dois elementos essenciais: um conjunto de constantes e um conjunto de operações básicas sobre estes valores.

Na maioria das linguagens funcionais, conhecendo-se o tipo dos valores da expressão podemos determinar o tipo do valor que dela resultará, ou seja, o seu contra-domínio. Em linguagens de programação isto equivale a dizer que a linguagem é ***fortemente tipada***.

Um tipo básico bastante importante são os números. Aqui nos interessa subdividi-los em inteiros e reais. Antes de irmos além, é importante que se saiba que, sendo o computador composto de elementos finitos, algumas adaptações e simplificações precisam ser feitas para o tratamento de números. Portanto ao invés de falarmos dos números inteiros, falaremos do tipo ***Integer*** e ao invés dos números reais, falaremos do tipo ***Real***.

Dizemos ainda que os tipos são elementares ou estruturados. Os numéricos são elementares, assim como também o são os tipos lógico e os caracteres.

5.2. O TIPO INTEGER: Comecemos pelo conjunto de valores. Enquanto números inteiros é um conjunto infinito, variando suas constantes de menos infinito a mais infinito, o tipo Integer (Int) trata com um conjunto de constantes finito, que pode variar em cada implementação. Por exemplo, podemos ter [- 32767 .. 32768].

O conjunto de operações pode variar, entretanto, em geral são fornecidos os seguintes:

`+`	adição
`*`	multiplicação
`-`	subtração
`//`	divisão inteira
`**`	potência
`%`	resto
`divmod`	quociente e resto de uma divisão
`abs`	valor absoluto

Exemplos:

```
>>> 2+3
5
>>> 5//2+8
10
```

5.3. O TIPO FLOAT: A denominação em Python para a adequação dos reais aos computadores, é denominada de ***Float***. Para este tipo não falaremos de uma lista de constantes. Ao invés disso falaremos aqui em magnitude e precisão de um número. Como na notação científica. A precisão nos diz quantos algarismos significativos são usados e a magnitude nos diz qual o maior expoente admitido.

Por exemplo, uma determinada implementação pode utilizar 6 algarismos significativos. Nesta, o número 123456.789 seria representado pelo número 123457.0, onde o 6 foi arredondado para 7.

Entretanto, a magnitude permite que números bem pequenos e bem grandes sejam representados, conforme exemplo abaixo, utilizando uma determinada implementação.

```
>>> 0.00000000000000000000000000000001
1e-36
>>> 10000000000000000000000000000000000.0
1e+37
>>> 0.00000000000000000000000000000000123456789987654321123456
1.234567899876543e-32
>>> 1.0e+37
1e+37
>>> 1.0e-38
1e-38
>>> 1.0e-39
1e-39
>>> 1.0e+39
1e+39
>>> 1.0e+59
1e+59
>>> 1.0e+99
1e+99
>>> 1.0e+999
inf
>>> 1.0e-999
0.0
>>> 1.0e+307
1e+307
>>> 1.0e+308
1e+308
>>> 1.0e+309
inf
>>> 1.0e-322
1e-322
>>> 1.0e-323
1e-323
>>> 1.0e-324
0.0
>>>
```

Nestes exemplos podemos observar que:

1. o expoente varia entre -323 a +308;
2. o uso de um expoente fora dos limites provoca aproximação para infinito ou zero;
3. podemos utilizar a notação científica.

O conjunto de operações aqui também pode variar, entretanto, em geral são fornecidos os seguintes (usualmente como parte de uma biblioteca de funções matemáticas):

`+`	Adição
`*`	Multiplicação
`-`	Subtração
`/`	Divisão
`**`	potência (o expoente tem que ser Int e positivo)
`trunc`	determina a parte inteira
`sin, cos, tan`	operações trigonométricas
`sqrt`	raiz quadrada
`log`	logaritmo na base ***e***
`log10`	logaritmo na base 10
`exp`	potência na base ***e***
`pi`	a constante 3.14159

Exemplos:

```
>>> sqrt(144)
12.0
>>> log(10)
2.302585092994046
>>> log10(10)
1.0
>>> pi
3.141592653589793
>>>
```

5.4. EXPRESSÕES MISTAS: A implementação corrente de Python permite que se construa expressões misturando constantes inteiras (***Int***) e reais (***Float***). Isto é feito considerando que os inteiros são subconjunto dos reais e portanto quando necessário substitui-se o inteiro pelo seu representante no conjunto dos reais. A

nova expressão, entretanto é considerada real, e, portanto o valor resultante será do tipo ***Float***.

Exemplos:

```
>>> 2+3
5
>>> 2+3.5
5.5
>>> (2+3.5)*2
11.0
>>> trunc(2+3.5)*2
10
>>>
```

5.5. PRECEDÊNCIA DOS OPERADORES: Quando aparecem vários operadores juntos na mesma expressão, certas regras de precedência são estabelecidas para resolver possíveis ambigüidades. A ordem em que os operadores são considerados é a seguinte:

1ª) `abs`, `sqrt` e qualquer outra função

2ª) `**`

3ª) `-x` `+x`

4ª) `*` `/` `//` `%`

5ª) `+` `-`

Da mesma forma que na matemática usual podemos usar os parêntesis para forçar uma ordem diferente de avaliação.

Exemplos:

```
>>> 2+3*5
17
```

*comentário: o operador * tem precedência sobre o operador +. Podemos escrever a expressão a seguir para denotar a sequência de avaliações:*

```
2 + 3 * 5 Þ 2 + 15 Þ 17
```

```
>>> (2+3)*5
25
```

comentário: a ordem de avaliação foi alterada pelo uso dos parêntesis. A sequência de avaliação é portanto:

```
(2 + 3) * 5 Þ 5 * 5 Þ 25
```

```
>>> 3*10%4+5
7
```

*comentário: a primeira prioridade é para avaliação do operador *, seguida da avaliação do operador % e finalmente do operador +. Podemos escrever a seguinte sequência de avaliação:*

```
3 * 10 % 4 + 5 Þ 30 % 4 + 5 Þ 2 + 5 Þ 7
```

```
>>> 3**10%4
1
```

*comentário : A primeira avaliação é do operador ** e em seguida é avaliado o operador %. A seqüência de avaliação é*

```
3 ** 10 // 4 Þ 59049 % 4 Þ 1
```

```
>>> 4**((20//4)%2)
4
```

comentário: A seqüência de avaliação é

```
4**((20//4)%2) Þ 4**(5%2) Þ 4**1 Þ 4
```

5.6. ORDEM DE ASSOCIAÇÃO: Quando há ocorrência de operadores de mesma precedência leva-se em consideração a ordem de associação que pode ser à direita ou à esquerda.

1. O operador de subtração faz associação à esquerda,

   5-4-2 = (5 - 4) - 2 e não 5 - (4 - 2)

2. Já o operador de exponenciação faz associação à direita,

   3**4**5 = 3 ** (4 ** 5) e não (3 ** 4) ** 5

O uso adequado das noções de precedência e associação serve também para escrevermos expressões com economia de parêntesis.

Observações:

1. A exponenciação quando repetida em uma expressão é avaliada da direita para a esquerda.
    1. `2**3**3` = 2**(3**3).
2. Os demais operadores, na situação acima, são avaliados da esquerda para a direita.
    1. `2 - 3 - 5` = (2 - 3) - 5
    2. `2 + 3 + 3` = (2 + 3) + 3

Exemplos:

```
>>> 3-7-2
-6
>>> 3*7+4
25
>>> 3*(7+4)
33
>>> 3**(1+3)
81
```

6. Expressões Lógicas e o Tipo Boolean

6.1. INTRODUÇÃO: Uma característica fundamental dos agentes racionais é a capacidade de tomar decisões adequadas considerando as condições apresentadas pelo contexto onde está imerso. Uma máquina que sabe apenas fazer contas, ou seja, manusear as operações aritméticas, terá sua utilidade fortemente reduzida e portanto não despertará tantos interesses práticos. Os computadores que temos hoje são passíveis de serem programados para tomada de decisão, ou seja, é possível escolher entre 2 ou mais ações, qual a que se deseja aplicar em um determinado instante. Em nosso paradigma de programação precisamos de dois elementos fundamentais: um tipo de dados para representar a satisfação ou não de uma condição e um mecanismo que use essas condições na escolha de uma definição. Neste capítulo discutiremos a natureza das proposições lógicas, sua aplicabilidade na resolução de problemas e introduziremos um novo tipo de dados, denominado de boolean. Satisfazendo logo de saída a curiosidade do leitor lembramos que o nome é uma homenagem a George Boole que estudou e formalizou as operações com estes tipos de valores.

6.2. PROPOSIÇÕES LÓGICAS: Revirando o baú das coisas estudadas no ensino fundamental, chegaremos às sentenças matemáticas. Estas são afirmações que se referem a conceitos matemáticos, tais como:

1. vinte e dois é maior que cinco
2. dois mais três é igual a oito
3. todo número primo é impar

O conceito de proposição lógica é mais geral, aplicando-se às mais diversas situações do cotidiano, como nos exemplo a seguir:

1. Maria é namorada de Pedro
2. José é apaixonado por Maria
3. Hoje é domingo

Analisando o significado da informação contida em uma proposição lógica podemos concluir que é uma afirmação verídica, quando ela se refere a fatos que realmente acontecem em um determinado mundo, ou, quando isso não ocorre, concluímos que elas são inverídicas. Obviamente que isto irá requerer que possamos ter acesso ao mundo considerado para podermos avaliar uma dada proposição.

Sentenças Abertas : as sentenças acima possuem uma característica importante: todos os seus personagens estão devidamente explicitados. Denominamos aquelas sentenças de sentenças fechadas. Uma sentença fechada pode ser

avaliada imediatamente. Um outro tipo de proposição importante são as sentenças abertas. Nestas, alguns personagens não estão devidamente explicitados e, portanto a sentença não pode ser avaliada. Quando tratamos sentenças abertas, antes é necessário instanciá-las para algum valor. Por exemplo, a sentença matemática,

x + 5 > 10

nem sempre é verdadeira, depende do valor que atribuirmos à variável ***x***. Quando atribuímos valores às variáveis de uma sentença dizemos que estamos instanciando a sentença.

No caso acima, podemos instanciar a sentença para os valores 3 e 10, entre outros, obtendo as seguintes instâncias:

1. 3 + 5 > 10
2. 10 + 5 > 10

Agora podemos avaliá-las e concluir que a segunda é verídica e a primeira não.

O mesmo poderia ocorrer com afirmações de outra natureza, como por exemplo:

Um certo cidadão está satisfeito com o Windows

Se substituirmos ***Um certo cidadão*** por Bill Gates, podemos concluir que a sentença é verídica. O mesmo não se pode concluir se a substituirmos pelo nome de inúmeras pessoas que arrancam os cabelos todos os dias, prejudicados que ficam com o mau funcionamento do citado programa.

Sentenças Compostas : O discurso do cotidiano e até o discurso matemático, pode ser escrito como uma lista de proposições simples. Como por exemplo:

1. Hoje é domingo
2. Aos domingos tem futebol
3. Quando meu time joga, eu assisto

Nem sempre isto é desejável ou suficiente. Para resolver a questão, há muito tempo atrás inventamos as sentenças compostas. Como por exemplo, para encurtar a escrita:

Três é menor que cinco e o quatro também.

Ou, em situações onde mais de uma proposição pode se aplicar, como em:

Domingo irei ao futebol ou escreverei notas de aula.

Ou ainda, quando queremos falar de situações que não ocorrem, como em:

Esperanto não é uma linguagem de programação.

Observamos então, o surgimento de três palavras para criar essas sentenças e que essas palavras (***e***, ***ou***, ***não***) não se referem a coisas, propriedades ou fenômenos de um determinado universo. Elas são denominadas de palavras lógicas, visto que a sua função na linguagem é possibilitar a articulação entre as proposições do discurso.

A composição de uma proposição pode envolver várias palavras lógicas, como nos exemplos a seguir:

1. ***Dois é menor que três e maior que um mas não é maior que a soma de três com dois;***
2. ***Maria gosta de Pedro e de Joana mas não gosta de Antonio.***

Avaliação de Sentenças Compostas: Para avaliar sentenças simples, vimos que bastava olhar para o mundo ao qual ela se refere e verificar se a situação expressa pela sentença ocorre ou não. E como proceder para as sentenças compostas? Um caminho que parece confiável é apoiar essa avaliação no papel representado por cada uma das palavras lógicas.

Assim, quando ao considerarmos a proposição

Hoje fui ao cinema e ao teatro,

só poderemos dizer que ela é verídica se tanto a proposição ***Hoje fui ao cinema*** quanto a proposição ***Hoje fui ao teatro*** forem avaliadas como verídicas.

Para a proposição composta

Maria foi a missa ou ao cinema,

devemos considerá-la como verídica se uma das duas proposições:

1. ***Maria foi a missa***
2. ***Maria foi ao cinema***

forem avaliadas como verídica. E se as duas forem avaliadas como verídicas? No discurso cotidiano tendemos a pensar nesta situação como inverídica visto que queríamos uma ou outra. A linguagem natural não estabelece se devemos ou não explicitar que não estamos falando do caso em que ambas podem ser constatadas. Podemos assumir, portanto, que a nossa sentença composta usando ***ou*** será avaliada como verídica quando pelo menos uma das duas proposições que a compõe for avaliada como verídica.

No caso da sentença composta usando a palavra lógica ***não***, como em

Hoje não choveu

diremos que ela será avaliada como verídica quando a proposição ***Hoje choveu*** não puder ser constatada e como inverídica no caso oposto.

6.3. O TIPO DE DADOS BOOLEAN: Podemos agora discutir sobre a natureza do valor resultante da avaliação de uma sentença. A avaliação é de natureza funcional, ou seja, dada uma sentença ***s*** ela será mapeada em um de dois valores distintos. Então, o contradomínio de uma avaliação é um conjunto com apenas dois valores, um para associar com avaliações verídicas e outras com as inverídicas. Podemos escolher um nome para esses valores. Historicamente, as linguagens de programação os tem denominado de ***True*** e ***False***. O primeiro para associar com as proposições que podem ser constatadas no mundo considerado e a segunda com as não constatáveis.

O tratamento das proposições compostas pode também receber um tratamento formal. Cada uma das palavras lógicas associamos uma função matemática cujo domínio e contradomínio será o conjunto que acabamos de definir com as constantes ***True*** e ***False***.

Em Python nossos operadores são representados por:

Palavra lógica	Símbolo em Python
e	`and`
ou	`or`
não	`not`

Vamos então formalizar a avaliação de sentenças compostas.

Sejam **s1** e **s2** duas proposições lógicas,

1. O valor lógico da sentença `s1 and s2` é ***True*** se e somente se o valor lógico de `s1` é ***True*** e o valor lógico de `s2` é ***True*** e é ***False*** em caso contrário.
2. O valor lógico da sentença `s1 or s2` é ***False*** se e somente se o valor lógico de `s1` é ***False*** e o valor lógico de `s2` é ***False*** e é ***True*** em caso contrário.
3. O valor lógico da sentença `not s1` é ***True*** se e somente se o valor lógico de `s1` é ***False*** e é ***False*** em caso contrário

Tabela Verdade: Estas definições também podem ser representadas através das de uma enumeração de suas associações, formando o que costuma-se chamar de ***Tabelas Verdade*** as quais apresentamos a seguir.

s1	s2	s1 and s2
True	True	True
True	False	False
False	True	False
False	False	False

s1	s2	s1 or s2
True	True	True
True	False	True
False	True	True
False	False	False

s1	not s1
True	False
False	True

6.4. OPERADORES RELACIONAIS: Quando falamos de proposições lógicas no discurso matemático, ou seja, sentenças matemáticas, usamos termos como: “menor do que”, “menor ou igual”, “diferente”, entre outros. Tais recursos são fundamentais para a nossa intenção de prover os computadores com capacidade de “decidir”. Denominados estes elementos de operadores relacionais, pois estabelecem uma relação de comparação entre valores de um mesmo domínio. O contradomínio deles é do tipo Boolean.

operador	*v1* op *v2*
==	v1 é igual a v2
!=	v1 é diferente de v2
<>	v1 é diferente de v2
<	v1 é menor que v2
<=	v1 é menor ou igual a v2
>	v1 é maior que v2
>=	v1 é maior ou igual a v2

Podemos usar estes operadores para construir novas definições ou simplesmente, em uma sessão de Hugs para solicitar a avaliação de uma expressão, como apresentado a seguir.

```
>>> 5>4
True
>>> 4!=(5+3)
True
>>> 5%2==15%2
True
>>> 5//2<=5/2
True
```

6.5. EXPRESSÕES E DEFINIÇÕES: Agora que temos um novo tipo de dados, ou seja, um par composto de constantes e funções, podemos pô-lo a nosso serviço, escrevendo expressões, de forma tão natural quanto aquela que usamos para escrever expressões aritméticas. Usando essas expressões podemos então construir definições cujo tipo resultante seja booleano. Os ingredientes básicos para construir essas expressões são os operadores relacionais.

Expressões simples : Por exemplo, para construir uma definição que avalie se um número é par, podemos usar a seguinte definição:

```
def par(x): return (x%2)==0
```

E que, quando usada, produzirá os resultados ilustrados abaixo:

```
>>> par(5)
False
>>> par(148)
True
>>> par(0)
True
```

Outros exemplos:

verificar se ***a*** é múltiplo de ***b***	`def multiplo(a,b):` `  return (a%b)==0`
verificar se ***a*** é divisor de ***b***	`def divisor(a,b):` `  return multiplo(a,b)`
verificar se uma distância ***d*** é igual à diagonal de um quadrado de lado ***a***	`def diag(d,a):` `  return (a*sqrt(2))==d`
verificar se um número é um quadrado perfeito	`def quadp(n):` `  return trunc(sqrt(n))==sqrt(n)`
verificar se dois números ***a*** e ***b*** são termos consecutivos de uma P.A. de razão ***r***	`def spa(a,b,r):` `  return (a+r)==b`

Expressões compostas : Podemos usar agora os operadores lógicos para construir expressões compostas. Como nos exemplos a seguir:

verificar se 3 números estão em ordem crescente	`def ordc(a,b,c):` `  return (a>b) and (b>c)`

verificar se um número ***x*** está no intervalo fechado definido por ***a*** e ***b***	`def pert(x,a,b):`
	`  return (x>=a) and (x<=b)`
	`# ou`
	`def pert2(x,a,b):`
	`  return not((x<a) or (x>b))`

Verificar se um determinado ponto do espaço cartesiano está no primeiro quadrante	`def pquad(x,y):` `  return (x>0) and (y>0)`

verificar se 3 números ***a***, ***b*** e ***c***, são lados de um triângulo retângulo	`def tret(a,b,c) :`
	`  return (`
	`  (a**2+b**2)==c**2 or`
	`  (a**2+c**2)==b**2 or`
	`  (b**2+c**2)==a**2 )`

Quando nossas expressões possuem mais de um operador lógico devemos observar a precedência de um sobre o outro. Por exemplo, na expressão

$$P \texttt{ or } Q \texttt{ and } R$$

onde P, Q e R são expressões lógicas, o operador `and` será avaliado primeiro pois tem precedência sobre o `or`, portanto a expressão será avaliada para True se P for avaliado para True ou se a sub-expressão (Q `and` R) for avaliada para True.

Podemos modificar esta ordem utilizando parêntesis como nas expressões aritméticas. A expressão acima poderia ser reescrita como:

$$\texttt{(} P \texttt{ or } Q \texttt{) and } R$$

E agora, para que ela seja avaliada para verdadeira é preciso que R seja avaliada para verdadeira.

Vejamos um exemplo:

Verificar se x está fora do intervalo definido por a e b e a e b estão em ordem não decrescente	`def f(x,a,b) :`
	`  return  (`
	`  ((x<=a) or (x>=b)) and (a<=b) )`

Verificar se x é menor que a ou se x é maior que b e a e b estão em ordem não decrescente	`def g(x,a,b):`
	`  return  (`
	`  ((x<a) or (x>b)) and (a<=b) )`

6.6. RESOLVENDO UM PROBLEMA: Desejamos verificar se um determinado ponto do espaço cartesiano está dentro ou fora de um retângulo paralelo aos eixos, conhecidos o ponto, o canto superior esquerdo e o canto inferior direito do retângulo.

Etapa 1 [Entendendo o problema] O ponto pode estar em qualquer quadrante? E o retângulo, pode estar em qualquer quadrante? Basta ter os dois cantos para definir o retângulo? Em que ordem serão informados os cantos? Como se descreve um canto? Um ponto que esteja sobre um dos lados, está dentro ou fora do retângulo?

Vamos assumir então as seguintes decisões: discutiremos inicialmente apenas sobre pontos e retângulos localizados no primeiro quadrante. A ordem em que os dados serão informados será: primeiro o ponto, depois o canto superior esquerdo e por ultimo o canto inferior direito. Para cada ponto serão informadas as duas coordenadas, primeiro a abscissa e depois a ordenada. Os pontos na borda são considerados pertencentes ao retângulo.

Etapa 2 [Planejando a Solução]: Conheço algum problema parecido? Posso decompor este problema em problemas mais simples? Sei resolver um problema mais geral em que este é um caso particular?

Bom, já nos envolvemos com o problema para verificar se um ponto estava num intervalo linear, este também se refere a intervalo. Verificar se um ponto pertence a uma região qualquer do espaço n-dimensional é mais geral, se eu conhecesse uma definição para este problema geral, bastaria instanciá-la. Será que posso decompor o problema na verificação de dois espaços lineares, um definido pelos lados paralelos ao eixo das ordenadas e outro paralelo ao eixo das abscissas? Se afirmativo, como combino as duas soluções?

Para que um ponto esteja dentro do retângulo é necessário que sua ordenada esteja entre as ordenadas dos dois cantos. Sabemos também que a abscissa precisa estar entre as abscissas dos dois cantos. Isso basta? Para combinar as soluções percebemos que é suficiente que as duas primeiras estejam satisfeitas.

Etapa 3 [Construindo a Solução] Construindo a solução - Anteriormente já elaboramos a definição de pertinência a um intervalo linear, vamos usá-la aqui.

`def pert(x,a,b):` `  return (x>=a) and (x<=b)`	pertinência linear
`def pertsup(x,y,x1,y1,x2,y2):` `  return pert(x,x1,x2) and pert(y,y2,y1)`	pertinência de superfície

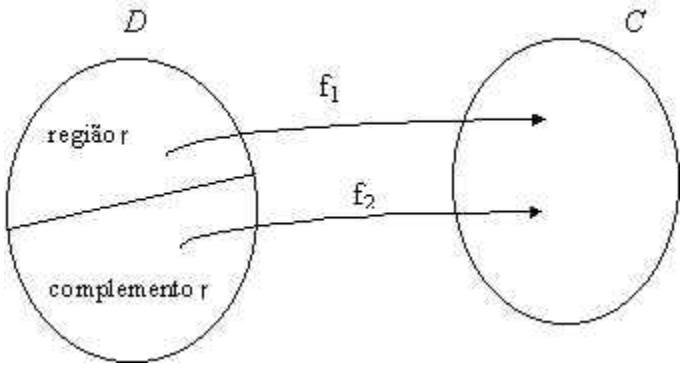
E o teste, para que valores interessam testar?

Etapa 4 [Analisando a Solução]: Existem outras maneiras de resolver o problema? Esta solução se aplica as outras dimensões?

7. Definições Condicionais

7.1. INTRODUÇÃO: Sabemos de nosso conhecimento matemático que algumas funções não são contínuas em um domínio e que, portanto, possuem várias definições.

Em muitos casos, o domínio *D* de uma função está dividido em regiões disjuntas que se complementam e, para cada uma dessas regiões, existe uma expressão que define o seu mapeamento no contra-domínio *C*. Podemos representar esta situação pela figura abaixo:



Exemplo 1 - Considere a função que determina o valor da passagem aérea de um adulto para um determinado trecho, considerando também a idade do comprador. Pessoas com idade a partir de 60 anos possuem um desconto de 40% do valor original. Considere ainda que a passagem para o trecho considerado custa R$ 600,00.

Temos aqui duas formas de calcular o valor da passagem de uma pessoa, dividindo o domínio em dois subconjuntos. O dos adultos com menos de 60 e o dos demais.

Podemos definir as duas funções:

```
def vpass1(): return 600
def vpass2(): return vpass1()*0.6
```

Para usa-las, temos que explicitamente escolher a que se aplica ao nosso caso.

Exemplo 2 - Considere a função que associa com um determinado rendimento o Imposto de Renda a ser pago. Até um determinado valor, o contribuinte não paga imposto, e a partir de então o rendimento é dividido em faixas (intervalos), aos quais se aplicam diferentes taxas. Suponha a tabela hipotética abaixo.

Faixa	alíquota	Desconto
inferior ou igual a 22.499,13	0	0,00
entre 22.499,14 e 33.477,72	7,5	1.687,43
entre 33.477,73 e 44.476,74	15	4.198,26
entre 44.476,75 e 55.373,55	22,5	7.534,02
acima de 55.373,55	27,5	10.302,70

Para descrever as várias definições e os correspondentes subdomínios, poderíamos escrever separadamente cada definição, construindo portanto várias funções, e deixar que o usuário escolha qual usar. Claro que isto traria muitos inconvenientes pois estaríamos deixando uma escolha “mecânica” na mão do usuário, que além de sobrecarregá-lo com tarefas desnecessárias, ainda estaria expondo-o ao erro por desatenção. Mas vamos lá construir as funções independentes.

```
def ir1(s): return 0
def ir2(s): return s*0.075-1687.43
def ir3(s): return s*0.15-4198.26
def ir4(s): return s*0.225-7534.02
def ir5(s): return s*0.275-10302.70
```

Agora, para usá-las, o usuário considera o seu salário, olha a tabela e seleciona qual função aplicar.

Podemos fazer melhor com expressões condicionais, vamos deixar que o computador escolha. Descrevemos cada subdomínio com a respectiva função aplicável e deixemos que ele escolha a definição a aplicar, dependendo do valor fornecido.

7.2. A ESTRUTURA *IF*: Uma expressão condicional construída com ***if*** possui a seguinte sintaxe:

`if` <expressão lógica 1> **`:`** <expressão 1>

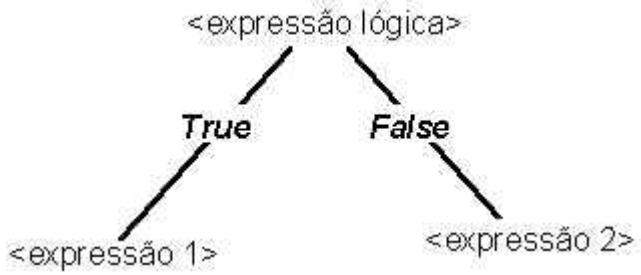
(**`elif`** <expressão lógica *i*> **`:`** <expressão *i*>)*

[**`else :`** <expressão *n*>]

onde:

<expressão lógica *i*>	Uma expressão descrevendo uma condição a ser satisfeita, envolvendo operadores relacionais e operadores lógicos.
<expressão *i*>	1. Expressões descrevendo um valor a ser produzido como resposta à entrada fornecida e, como a expressão total tem que ser de um único tipo, as expressões devem ser do mesmo tipo. 2. Cada uma destas expressões pode ser inclusive uma outra condicional, dentro da qual pode haver outras e assim sucessivamente. 3. Apenas uma das instâncias dessas expressões é executada

Árvore de decisão: Podemos representar este tipo de expressão em uma notação gráfica denominada de árvore de decisão. É importante considerar que este tipo representação é uma ferramenta importantíssima para estruturarmos a solução de problemas que requerem expressões condicionais.



Um bom primeiro exemplo é a definição de valor absoluto de um número, que possui a seguinte estrutura:

subdomínio	expressão
x < 0	-x
x >= 0	x

Como só temos duas possibilidades, podemos codificar da seguinte maneira:

```
def absoluto(x):
  if x<0 : return -x
  else : return x
```

No Exemplo 1, do problema do preço da passagem aérea, podemos definir as seguintes funções, onde p é a idade do passageiro.

```
def vpass1(): return 600
def vpass2(p):
  if p<60 : return vpass1()
  else : return vpass1()*0.6
```

Para o problema de nosso Exemplo 2, o domínio deve ser quebrado em cinco subdomínios e para cada um deles construiremos uma expressão.

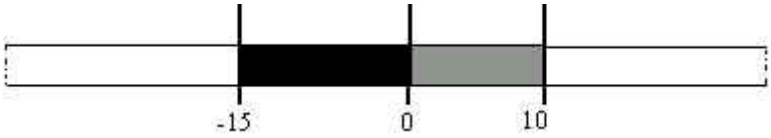
domínio	função
s ≤ 22.499,13	`ir1(s)`
22.499,14 ≤ s ≤ 33.477,72	`ir2(s)`
33.477,73 ≤ s ≤ 44.476,74	`ir3(s)`
44.476,75 ≤ s ≤ 55.373,55	`ir4(s)`
s > 55.373,55	`ir5(s)`

Para codificar precisaremos ir quebrando sucessivamente o domínio da função. Primeiro dividimos entre o primeiro intervalo e o resto, depois dividimos o resto entre o segundo intervalo e o resto e assim sucessivamente.

A codificação final pode ser:

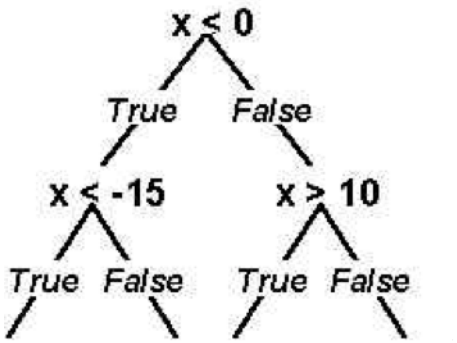
```
def ir(s):
  if s<=22499.13 : return ir1(s)
  elif pert(s,22499.14,33477.72) : return ir2(s)
  elif pert(s,33477.73,44476.74) : return ir3(s)
  elif pert(s,44476.75,55373.55) : return ir4(s)
  else : return ir5(s)
```

7.3. USANDO O IF - EXEMPLO 1: Considere um mapeamento de valores numéricos onde o domínio se divide em 4 regiões, cada uma das quais possui diferentes formas de mapeamento. As regiões são apresentadas na figura abaixo, numeradas da esquerda para direita. Observe ainda que as extremidades são abertas. Considere ainda a seguinte tabela de associações:



região	mapeamento desejado
região 1	o dobro de x
região 2	o sucessor de x
região 3	o quadrado de x
região 4	o simétrico do quadrado de x

Podemos analisar as regiões através do seguinte diagrama:

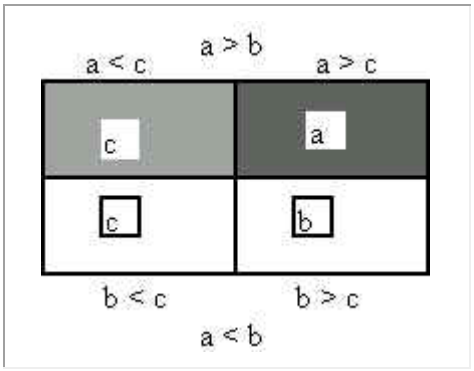


O que nos levará à seguinte definição em Python:

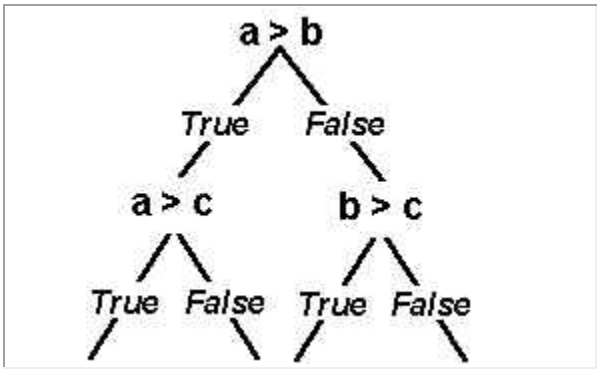
```
def ex1(x):
  if x<0 :
    if x<(-15) : return (2*x)
    else : return (x+1)
  elif x<10 : return (x**2)
  else : return -(x**2)
```

7.4. USANDO O IF - EXEMPLO 2: Dados 3 números inteiros distintos, determinar o maior deles.

Podemos explorar uma solução da seguinte maneira. Considere um retângulo e divida-o horizontalmente em 2 partes, a parte de cima representa as situações onde a > b e a de baixo aquelas onde b > a. Divida agora o retângulo verticalmente, em cada uma das regiões anteriores surgirão 2 metades. Na de cima, representamos agora a relação entre a e c. Na de baixo, a relação entre b e c.



Explorando as relações entre os números



Representando as relações através de uma árvore de decisão

Traduzindo a árvore de decisão para Python, chegamos à seguinte definição:

```
def maior(a,b,c):
  if a>b :
    if a>c : return (a)
    else : return (c)
  elif b>c : return (b)
  else : return (c)
```

8. O Teste de Programas

8.1. INTRODUÇÃO: Não basta desenvolver o programa para resolver um dado problema. É preciso garantir que a solução está correta. Muitos erros podem ocorrer durante o desenvolvimento de um programa e portanto temos que garantir que o programa a ser executado está livre de todos eles. Ao conceber a solução podemos nos equivocar e escolher caminhos errados, precisamos então eliminar esses equívocos. Ao codificarmos a nossa solução podemos cometer outros erros ao não traduzirmos corretamente nossa intenção. Esses erros podem ocorrer por um mau entendimento dos elementos da linguagem ou até mesmo por descuido, o certo é que eles ocorrem. Uma ferramenta utilíssima, embora não infalível, é o teste de programa.

Em sua essência, o teste de programa consiste em submeter um programa ao exercício de algumas instâncias do problema e comparar os resultados esperados com os resultados obtidos.

8.2. O PROCESSO DE TESTE: Em primeiro lugar devemos escolher as instâncias apropriadas, pois não basta escolhê-las aleatoriamente. A seguir devemos determinar, sem o uso do programa, qual o valor que deveria resultar quando o programa for alimentado com essas instâncias. O passo seguinte consiste em submeter cada instância ao programa e anotar o resultado produzido por ele. Finalmente devemos comparar cada valor esperado com o valor produzido e descrever qual o tipo de ocorrência.

Um exemplo: Considere o problema de identificar se um dado ponto está ou não localizado no primeiro quadrante do espaço cartesiano. Considere ainda que obtivemos a seguinte definição:

```
def primquad(x,y) : return (x>=0 and y>=0)
```

Precisamos agora verificar se ela atende nossa intenção. Para tanto devemos escolher algumas instâncias, prever o resultado esperado e em seguida submeter ao Python, para ver o que acontece.

Que pares de valores deveremos escolher? Bom, vamos escolher uns pares usando a seguinte estratégia: um par onde x é maior que y, outro onde y seja maior que x e um terceiro em que os dois sejam iguais. Gerando uma planilha como apresentada a seguir.

x	y	resultado esperado	resultado obtido	diagnóstico
-5	-2	False		
-2	-5	False		
5	5	True		

Podemos agora submeter as instâncias à avaliação do sistema, obtendo a seguinte interação:

```
>>> primquad(-5,-2)
False
>>> primquad(-2,-5)
False
>>> primquad(5,5)
True
```

Podemos agora completar o preenchimento de nossa planilha, obtendo a tabela a seguir:

x	y	resultado esperado	resultado obtido	diagnóstico
-5	-2	False	`False`	sucesso
-2	-5	False	`False`	sucesso
5	5	True	`True`	sucesso

Muito bem, nosso programa está correto. Passou com sucesso em todos os testes!

Que pena que não seja verdade, apesar de passar em todos os testes a que foi submetido, ele não funciona corretamente. Tudo que podemos afirmar neste instante é que para os valores usados, o programa funciona corretamente. E para os outros valores, será que funciona corretamente?

Outros valores? Quais?

8.3. PLANO DE TESTE: Para escolher os valores que usaremos, antes de mais nada devemos identificar as classes de valores que serão relevantes para o teste, em um segundo instante podemos então escolher os representantes destas classes. Quando temos um parâmetro, os possíveis valores a serem usados são todas as constantes do domínio. Para o caso de um parâmetro do tipo inteiro, existem pelo menos 65536[1] valores diferentes. Testar nosso programa para todos esses valores implicaria em determinar a mesma quantidade de resultados esperados e em seguida digitar e submeter esta mesma quantidade de instâncias do problema para o sistema e ainda depois conferir um a um os resultado obtidos com os esperados. Enfim, um trabalho imenso. Imagine agora se fossem dois parâmetros? A quantidade de pares seria da ordem de 4.29497 x 10^09 (algo da

[1] Maior valor inteiro aceito nos interpretadores de linguagens dos primeiros computadores.

ordem de quatro bilhões). E para 3 parâmetros? E para ***n*** parâmetros? Números cada vez mais enormes. Por este caminho, testar programas seria inviável.

Felizmente não precisamos de todos eles, basta identificar as classes distintas que importam para o problema, ou seja, as classes de equivalência relevantes. Isto pode ser obtido analisando o enunciado estendido do problema.

No caso de nosso exemplo anterior, analisando melhor a definição, podemos identificar que por definição o espaço cartesiano se divide em quatro regiões. A primeira, onde ambos as coordenadas são positivas, denominamos de primeiro quadrante. A segunda, onde a coordenada x é negativa e a y positiva, denominamos de segundo quadrante. O terceiro quadrante é o espaço onde ambas as coordenadas são negativas. Ainda temos o quarto quadrante onde a coordenada x é positiva e a y é negativa. E os pontos que ficam em um dos eixos ou na interseção destes, qual a classificação que eles têm? Bom, podemos convencionar que não estão em nenhum dos 4 quadrantes descritos. Resumindo, nosso plano de teste deve contemplar estas situações, conforme ilustra-se na tabela a seguir.

x	**y**	**quadrante**
positivo	positivo	primeiro
negativo	positivo	segundo
negativo	negativo	terceiro
negativo	positivo	quarto
nulo	qualquer não nulo	eixo das ordenadas
qualquer não nulo	nulo	eixo das abscessos
nulo	nulo	origem

É bom observar ainda que este plano pode e deve ser preparado antes mesmo de elaborar o programa, para fazê-lo, precisamos apenas da especificação detalhada. Além disso, este plano não precisa ser feito pelo programador responsável pela elaboração da solução, qualquer outro programador, de posse do enunciado detalhado, pode se encarregar da tarefa. Este tipo de plano serve para alimentar o teste denominado de *“caixa preta”*. Esta denominação se deve ao fato de não ser necessário conhecermos a estrutura da solução para elaborar o plano. Um outro aspecto positivo da elaboração do plano o mais cedo possível é que isso contribui para um melhor entendimento do problema.

Voltando ao nosso exemplo, podemos agora elaborar a nossa planilha de testes considerando as classes de equivalência a serem definidas. Uma questão que surge é como escolhemos o representante de uma classe? Existem melhores e piores? No nosso caso, como pode ser qualquer valor positivo ou negativo,

podemos escolher valores de um dígito apenas, no mínimo escreveremos e digitaremos menos.

x	y	resultado esperado	resultado obtido	diagnóstico
2	3	True		
-2	3	False		
-2	-3	False		
2	-3	False		
0	3	False		
0	-3	False		
2	0	False		
-2	0	False		
0	0	False		

8.4. REALIZANDO O TESTE: Vejamos agora o resultado de nossa interação com o sistema.

```
>>> primquad(2,3)
True
>>> primquad(-2,3)
False
>>> primquad(-2,-3)
False
>>> primquad(2,-3)
False
>>> primquad(0,3)
True
>>> primquad(0,-3)
False
>>> primquad(2,0)
True
>>> primquad(-2,0)
False
>>> primquad(0,0)
True
```

Voltemos agora para nossa planilha e vamos preenchê-la na coluna de resultado obtido e diagnóstico.

x	y	resultado esperado	resultado obtido	diagnóstico
2	3	True	`True`	sucesso
-2	3	False	`False`	sucesso
-2	-3	False	`False`	sucesso
2	-3	False	`False`	sucesso
0	3	False	`True`	falha
0	-3	False	`False`	sucesso
2	0	False	`True`	falha
-2	0	False	`False`	sucesso
0	0	False	`True`	falha

8.5. DEPURAÇÃO: Uma vez testado o programa e identificado que ocorreram instâncias para as quais a nossa definição não está fazendo a associação correta, ou seja, o valor obtido é diferente do esperado, devemos passar a uma nova fase. *"Depurar"* um programa é um processo que consiste em buscar uma explicação para os motivos da falha e posteriormente corrigi-la. Obviamente isto também não é um processo determinante e nem possuímos uma fórmula mágica. Ao longo de nossa formação de programadores iremos aos poucos incorporando heurísticas que facilitem esta tarefa. Por enquanto é muito cedo para falarmos mais do assunto e vamos concluir com um fechamento do problema anterior. Após concluir as modificações devemos testar o programa novamente.

Depurando nossa solução - Podemos concluir por simples inspeção da nossa última planilha (aquela com todas as colunas preenchidas) que nossa solução está incorreta. Uma interpretação dos resultados nos leva à hipótese de que a nossa solução considera que quando o ponto se localiza na origem ou em um dos eixos positivos, a nossa definição está considerando que eles estão no primeiro quadrante.

Passo seguinte, verificar se de fato nossa definição incorre neste erro. Em caso afirmativo, corrigi-la e a seguir resubmetê-la aos testes.

Observando a nossa definição inicial, podemos concluir que de fato nossa hipótese se confirma.

```
def primquad(x,y) : return (x>=0 and y>=0)
```

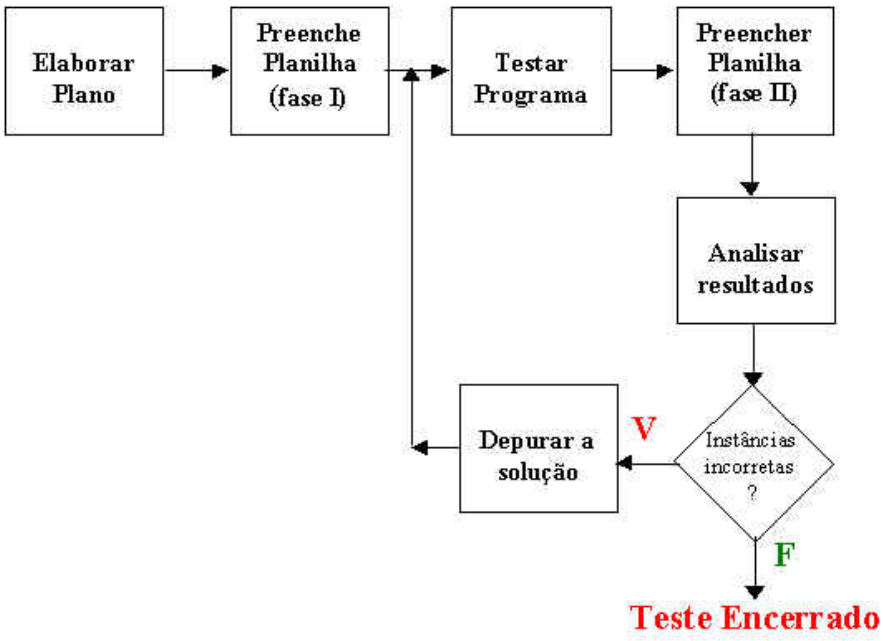
Podemos então modificá-la para obter uma nova definição, que esperamos que esteja correta.

```
def primquad(x,y) : return (x>0 and y>0)
```

Agora é submetê-la novamente ao teste e ver o que acontece!

8.6. UMA SÍNTESE DO PROCESSO: O processo é, como vimos, repetitivo e só se encerra quando não identificarmos mais erros. O diagrama ilustra o processo como um todo.

Mas lembre-se, isso ainda não garante que seu programa esteja 100% correto! Quando não identificamos erro, apenas podemos concluir que para as instâncias que usamos o nosso programa apresenta os resultados esperados.



9. Resolvendo Problemas - Os Movimentos do Cavalo

9.1. INTRODUÇÃO: Considere o jogo de xadrez, onde peças são movimentadas em um tabuleiro dividido em 8 linhas e oito colunas. Considere ainda os movimentos do cavalo, a partir de uma dada posição, conforme diagrama a seguir, onde cada possível movimento é designado por ***mi*.** No esquema, o cavalo localizado na posição (5,4) pode fazer oito movimentos, onde o primeiro deles, ***m1***, levaria o cavalo para a posição (7,5).

8								
7								
6				m3		m2		
5			m4				m1	
4					**C**			
3			m5				m8	
2				m6		m7		
1								
	1	2	3	4	5	6	7	8

9.2. PROBLEMA 1: Escreva uma função que determina se, a partir de uma dada posição, o cavalo pode ou não realizar o primeiro movimento. Vamos chamá-la de ***pmov***, e denominar seus parâmetros (a posição corrente), de ***x*** e ***y***. Eis alguns exemplos de uso de nossa função:

```
>>> pmov(5,4)
     True
>>> pmov(8,1)
    False
>>> pmov(1,1)
     True
>>> pmov(1,8)
    False
```

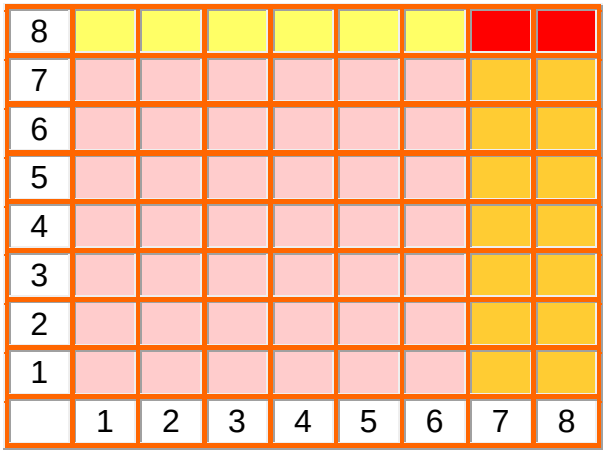
9.2.1. SOLUÇÃO - A solução pode ser encontrada através da construção de uma expressão booleana que avalie se a nova posição, ou seja, aquela em que o cavalo seria posicionado pelo primeiro movimento, está dentro do tabuleiro. Como o cavalo, no primeiro movimento, anda duas casas para direita e uma para cima,

basta verificar se as coordenadas da nova posição não ultrapassam a oitava fileira (linha ou coluna).

Codificando em Python, temos então:

```
def pmov(x,y): return ((x+2)<=8 and (y+1)<=8)
```

9.2.2. TESTANDO A SOLUÇÃO - Como já discutimos anteriormente, para verificar a correção de nossa solução, devemos submetê-la a um teste. Para tanto devemos escolher as classes de equivalências relevantes e, a partir delas, produzir a nossa planilha de testes. Olhando para a especificação do problema, podemos identificar 4 conjuntos de valores que se equivalem para os fins do nosso programa, como podemos observar na figura abaixo:



9.2.3. ESTENDENDO O PROBLEMA - Podemos fazer o mesmo para todos os demais movimentos, obtendo com isso as seguintes definições:

```
def pmov(x,y): return ((x+2)<=8 and (y+1)<=8)
def smov(x,y): return ((x+1)<=8 and (y+2)<=8)
def tmov(x,y): return ((x-1)>=1 and (y+2)<=8)
def qmov(x,y): return ((x-2)>=1 and (y+1)<=8)
def qtmov(x,y): return ((x-2)>=1 and (y-1)>=1)
def sxmov(x,y): return ((x-1)>=1 and (y-2)>=1)
def stmov(x,y): return ((x+1)<=8 and (y-2)>=1)
def omov(x,y) : return ((x+2)<=8 and (y-1)>=1)
```

9.2.4. IDENTIFICANDO ABSTRAÇÕES - Podemos agora indagar, olhando para as oito definições, sobre a ocorrência de algum conceito geral que permeie todas elas. Poderíamos também ter feito isso antes. Como não o fizemos, façamo-lo agora. Podemos observar que todas elas avaliam duas expressões e que ambas testam fronteiras que podem ser margem direita, margem esquerda, margem superior ou margem inferior. Podemos observar ainda, que o par margem direita e margem superior testam o mesmo valor 8, assim como ocorre com as duas outras, que testam o valor 1. Com isso podemos definir duas novas funções, ***f*** e ***g***, para testar estes limites. Agora, as nossas definições anteriores podem ser reescritas, usando as duas abstrações identificadas.

```
def pmov(x,y): return (f(x+2) and f(y+1))
def smov(x,y): return (f(x+1) and f(y+2))
def tmov(x,y): return (g(x-1) and f(y+2))
def qmov(x,y): return (g(x-2) and f(y+1))
def qtmov(x,y): return (g(x-2) and g(y-1))
def sxmov(x,y): return (g(x-1) and g(y-2))
def stmov(x,y): return (f(x+1) and g(y-2))
def omov(x,y) : return (f(x+2) and g(y-1))
def f(w): return w<=8
def g(w): return w>=1
```

9.2.5. ANÁLISE DA SOLUÇÃO - O que será que ganhamos com esta nova forma de descrever a nossa solução? Podemos indicar pelo menos três indícios de vantagem na nova solução:

1. Clareza - Na medida em que agora está explicitado, que todas as oito funções para verificar os movimentos possuem estrutura semelhante e que todas estão usando funções para verificar a ultrapassagem das bordas;
2. Manutenção - Se nosso tabuleiro mudasse, ou seja, passasse a ter 9 linhas por nove colunas, bastaria alterar a função ***f*** e tudo estaria modificado, ao invés de termos que alterar as oito definições.
3. Reuso - As duas funções que testam as bordas poderiam ser usadas para construir funções para avaliar o movimento de outras peças do jogo de xadrez.

9.3. PROBLEMA 2: Sabemos que para cada posição alguns movimentos podem ser realizados e outros não. Como ordenamos os movimentos no sentido anti-horário, gostaríamos de obter, para uma dada posição, dentre os movimentos que

podem ser realizados, aquele que possui o menor índice. Vejamos o esquema abaixo.

8	**m4**				**m1**			**C1**
7			**C3**			m5		
6	**m5**				**m8**		m6	
5		**m6**		**m7**				
4								
3		**m2**					**m3**	
2			**m1**			**m4**		
1	**C4**							**C2**
	1	2	3	4	5	6	7	8

Podemos observar que o cavalo C1 só pode fazer os movimentos m5 e m6 e que o de menor índice é m5. Já o cavalo C2 só pode fazer os movimentos m3 e m4 e que o de menor índice é o m3. Enquanto isso o cavalo C3 pode fazer os movimentos m1, m4, m5, m6, m7 e m8. Para este caso o movimento de menor índice é o m1.

Vamos chamar esta função de ***qualmov*** e, como no problema anterior, os parâmetros serão as coordenadas da posição atual do cavalo. Eis alguns exemplos de uso de nossa função:

```
>>> qualmov(8,1)
3
>>> qualmov(8,8)
5
>>> qualmov(3,7)
1
>>> qualmov(1,1)
1
```

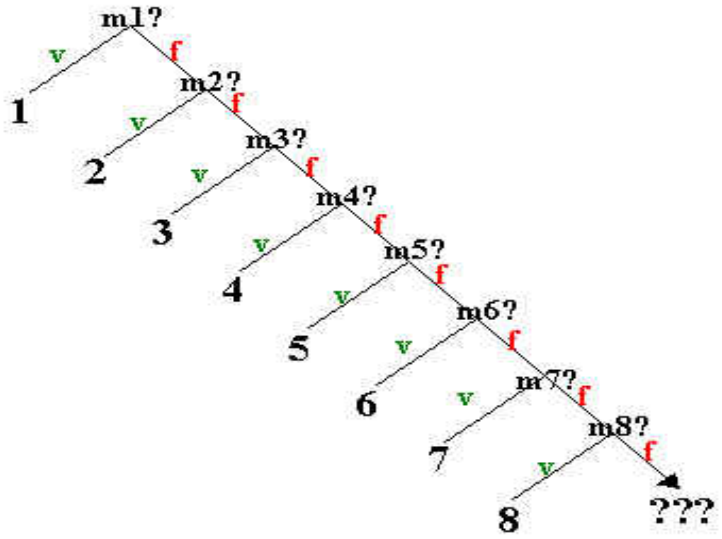
9.3.1. SOLUÇÃO - Bem, como já sabemos, para verificar se um dado movimento ***mi*** é possível, basta arranjar um meio de sair verificando um-a-um os movimentos, a partir do primeiro (m1) e encontrar o primeiro que pode ser realizado. Quando isso ocorrer podemos fornecer como resposta o seu índice. Podemos construir para isso uma árvore de decisão. Na raiz da árvore estará a pergunta

"é possível realizar o movimento m1"?

Em caso afirmativo (braço esquerdo da árvore), mapeamos o valor 1 e em caso negativo (braço direito), o que devemos fazer? Bom, aí podemos começar uma nova árvore (na verdade uma sub-árvore), em cuja raiz estará a pergunta:

"é possível realizar o movimento m2"?

E daí prosseguimos até que todos os movimentos hajam sido considerados. A árvore resultante será:



9.3.2. CODIFICANDO A SOLUÇÃO - Vamos então explorar os recursos da linguagem para transformar nosso plano em um programa que de fato possa ser "entendido" pelo nosso sistema de programação (Python). Como podemos observar temos aqui o caso de uma função que não é contínua para o domínio do problema. Pelo que sabemos até então, não dá para expressar a solução como uma única expressão simples. Resta-nos o recurso das expressões condicionais. Para verificar se um dado movimento é satisfeito podemos usar as funções que construímos anteriormente e com isso obtemos a seguinte definição:

```
def qualmov(x,y):
  if pmov(x,y): return 1
  elif smov(x,y): return 2
  elif tmov(x,y): return 3
  elif qmov(x,y): return 4
  elif qtmov(x,y): return 5
  elif sxmov(x,y): return 6
  elif stmov(x.y): return 7
  elif omov(x,y): return 8
  else : return 0
```

9.3.3. ANÁLISE DA SOLUÇÃO - Em primeiro lugar, incluímos a resposta igual a zero (0) quando o movimento m8, o último a ser avaliado, resulta em fracasso. Para que serve isso? Acontece que se a posição de entrada não for válida, ou

seja, uma ou ambas as coordenadas não pertencerem ao intervalo [1,8], nenhum movimento seria válido e se não providenciarmos uma resposta alternativa, nossa função seria parcial. Mas isto resolve de fato nosso problema? O que ocorreria se a posição de entrada fosse (0,0)? Bom, nossa função determinaria que o primeiro movimento poderia ser realizado e isto não é verdade. A invenção de um resultado extra para indicar que não há solução possível, transformando uma função parcial em uma função total, parece ser boa, mas como foi feita não resolveu. Em geral o melhor nestas situações é preceder toda e qualquer tentativa de determinar a solução adequada, por uma avaliação da validade dos dados de entrada. Neste caso, bastaria verificar se os dois estão no intervalo [1,8]. Vamos construir aqui uma função que avalia a pertinência de um valor a um intervalo numérico, conforme definição a seguir:

```
def pert(x,a,b): return (x>=a) and (x<=b)
```

Especulando um pouco mais sobre a nossa solução, podemos observar que o movimento ***m8***, jamais ocorrerá! Analisando os possíveis movimentos chegaremos à conclusão de que para nenhuma posição, o oitavo é o único movimento possível. Sugerimos fortemente que o leitor prove este teorema. Portanto a solução final pode ser:

```
def qualmov(x,y):
  if not(pert(x,1,8)) or not(pert(y,1,8)) : return 0
  elif pmov(x,y): return 1
  elif smov(x,y): return 2
  elif tmov(x,y): return 3
  elif qmov(x,y): return 4
  elif qtmov(x,y): return 5
  elif sxmov(x,y): return 6
  else : return 7
```

9.4. REVISITANDO O PROBLEMA 1: Observando a solução encontrada para o problema 1, constatamos que embora a noção de movimento do cavalo seja única, quem precisar saber se um dado movimento é válido, precisará conhecer o nome das oito funções. Embora seja cedo para falarmos de interface homem-máquina, já dá para dizer que estamos sobrecarregando nosso usuário ao darmos oito nomes para coisas tão parecidas. Será que temos como construir uma só função para tratar o problema? Vamos reproduzir aqui a interface das oito:

```
pmov(x,y)
smov(x,y)
tmov(x,y)
qmov(x,y)
```

```
qtmov(x,y)
sxmov(x,y)
stmov(x,y)
omov(x,y)
```

Propositadamente escrevemos o nome delas com um pedaço em vermelho e outro em preto (seria alguma homenagem a algum time que tem essas cores?). Na verdade estamos interessados em destacar que a pequena diferença nos nomes sugere que temos uma mesma função e que existe um parâmetro oculto. Que tal explicitá-lo? Podemos agora ter uma função com 3 parâmetros, sendo o primeiro deles para indicar o número do movimento que nos interessa. A interface agora seria:

```
mov(m,x,y)
```

Agora, por exemplo, para solicitar a avaliação do sétimo movimento para um cavalo em (3, 4), escrevemos:

```
>>> mov(7,3,4)
True
```

Muito bem, e como codificaremos isso?

9.4.1. SOLUÇÃO - Precisamos encampar em nossa solução o fato de que a nossa função possui diferentes formas de avaliação, para diferentes valores do domínio, algo parecido com a solução do problema 2, ou seja, a nossa função não é continua e portanto temos que selecionar qual a definição apropriada para um determinado valor de m. Devemos construir uma árvore de decisão. Aqui deixamos esta tarefa a cargo do leitor e passamos direto à codificação conforme apresentamos a seguir:

```
def mov(m,x,y):
  if not(pert(m,1,8)) : return False
  elif m==1 : return pmov(x,y)
  elif m==2 : return smov(x,y)
  elif m==3 : return tmov(x,y)
  elif m==4 : return qmov(x,y)
  elif m==5 : return qtmov(x,y)
  elif m==6 : return sxmov(x,y)
  elif m==7 : return stmov(x,y)
  else : return omov(x,y)
```

9.4.2. ANÁLISE DA SOLUÇÃO - Ao contrário da solução do problema 2, onde necessariamente a validade dos movimentos tinha que ser verificada do primeiro para o último, pois nos interessava saber qual o primeiro possível de ser realizado, o leitor pode observar que esta ordem aqui não é necessária. Tanto faz se perguntamos primeiro se m=7 ou se m=3. Será que podemos tirar algum proveito disso? Alertamos o iniciante, que devemos sempre identificar propriedades internas do problema e explorá-las adequadamente. Qual a influência desta ordem na eficiência da avaliação de uma expressão submetida ao Python? Para responder, basta lembrar que as condições são avaliadas sequencialmente. Por exemplo, se m=8, teremos que avaliar 8 condições, se m=1 faremos 2 avaliações e se m está fora do domínio faremos uma avaliação. Ou seja, no pior caso faremos 8 avaliações e no melhor caso uma (1). Em média, ao longo do uso da função, assumindo uma distribuição uniforme dos valores de m, faremos 4 avaliações. E se o intervalo fosse de 100 elementos distintos, quantas avaliações de condições faríamos em média? Será possível elaborar soluções onde este número de avaliações seja menor?

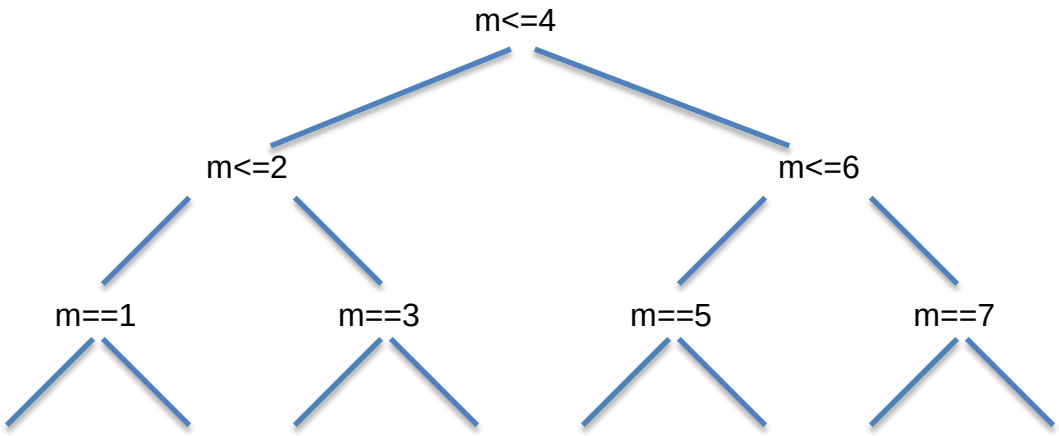
A resposta é sim! Já que a ordem das avaliações não importa, podemos buscar uma ordem mais conveniente para reduzir o número de avaliações por cada instância avaliada.

A ideia geral para estes casos é obtida a partir de um conceito de vasta utilidade na Computação. Falamos de ***árvore binária*** e o leitor por certo ouvirá falar muito dela ao longo da vida enquanto profissional da área.

O caminho consiste em dividir intervalo considerado ao meio e fazer a primeira pergunta, por exemplo, m<=4 ?

Dividindo em 2 o intervalo de comparações, para cada um destes podemos novamente dividir ao meio, até que não mais tenhamos o que dividir, como ilustramos no diagrama a seguir, onde cada linha representa um passo da divisão dos intervalos.

Que também podemos representar por:



A codificação ficará então:

```
def mov(m,x,y) :
  if not(pert(m,1,8)) : return False
  elif m<=4 :
    if m<=2 :
      if m==1 : return pmov(x,y)
      else : return smov(x,y)
    else :
      if m==3 : return tmov(x,y)
      else : return qmov(x,y)
  elif m<=6 :
    if m==5 : return qtmov(x,y)
    else : return sxmov(x,y)
  elif m==7 : return stmov(x,y)
  else : return omov(x,y)
```

Se fizermos uma análise das possibilidades veremos que para qualquer valor de m o número máximo de avaliações que faremos será sempre igual a quatro! E se fosse 100 valores para m, qual seria o número máximo de comparações? E para 1000? E para 1000000?

O número de comparações, seguindo este esquema, é aproximadamente igual ao logaritmo do número de valores na base 2. Portanto para 100 teríamos 7, para 1000 teríamos 10 e para um milhão teríamos 20. Veja que é bem inferior ao que obtemos com o esquema anterior, também denominado de linear. Confira a comparação na tabela a seguir.

número de valores	**esquema linear** (número médio)	**esquema binário** (número máximo)
8	8	4
100	50	7
1000	500	10
1000000	500000	20

10. Tuplas

10.1. INTRODUÇÃO: Até então trabalhamos com valores elementares. Ou seja, valores atômicos, indivisíveis a nível das operações de seu tipo de dados. Por exemplo, ***True***, ***523***, ***8.234***, são valores elementares dos tipos Boolean, Integer e Float, respectivamente. Não podemos, por exemplo, nos referir, dentro da linguagem, ao primeiro algarismo do valor ***523***. Dizemos que esses são tipos primitivos da linguagem. Além desses três, o Python provê ainda o tipo *Character*. O elenco de tipos primitivos de uma linguagem pode ser entendido como o alicerce da linguagem para a descrição de valores mais complexos, que são denominados genericamente de ***valores estruturados***.

Os problemas que pretendemos resolver estão em universos mais ricos em abstrações do que esse das linguagens. Para descrever esses mundos, precisamos usar abstrações apropriadas para simplificar nosso discurso. Por exemplo, quando quero saber onde alguém mora, eu pergunto: Qual o seu ***endereço***? Quando quero saber quando um amigo nasceu, eu pergunto: Qual a ***data*** do seu nascimento? E quando quero saber para onde alguém vai deslocar o cavalo no jogo de xadrez, eu pergunto: Qual a nova ***posição***? Os nomes *endereço*, *data* e *posição* designam valores estruturados. Uma data tem três partes: dia, mês e ano. Um endereço pode ter quatro: rua, número, complemento e bairro. Já a posição no tabuleiro tem duas partes, linha e coluna.

Para possibilitar a descrição de valores dessa natureza, o Python dispõe de um construtor denominado ***tupla***. Podemos definir uma tupla como um agregado de dados, que possui quantidade pré-estabelecida de componentes (dois ou mais), e onde cada componente da tupla pode ser de um tipo diferente (primitivo ou não).

10.2. DEFINIÇÃO DO CONCEITO: A representação de uma tupla é feita com a seguinte sintaxe:

(t1, t2, t3, ..., tn)

Onde cada ***ti*** é um termo da tupla.

Se ***Ti*** é o tipo de ***ti***, então o universo de uma tupla é dado por:

TT = T1 x T2 x ... x Tn

Sabendo-se que os conjuntos bases dos tipos ***Ti*** são ordenados, também os elementos de ***TT*** o serão.

Exemplos de Tuplas:

	Intenção	Representação
1	uma data	`(15, 05, 2000)`
2	uma posição no tabuleiro de xadrez	`(3, 8)`
3	uma pessoa	`("Maria Aparecida", "solteira", (3,02,1970))`
4	um ponto no espaço	`(3.0,5.2,34.5)`
5	uma carta de baralho	`(7, "espada")`
6		`(True, 3)`
7		`(False, 5//2)`
8		`(x, y, x+y)`

10.3. COMPONDO TUPLAS: Os exemplos apresentados já parecem suficientes para que tenhamos entendido como descrever um valor do tipo tupla. Vamos então apenas comentar sobre os exemplos e ilustrar o uso de tuplas nas definições.

Uma tupla é um valor composto. Isto significa que devemos colocar entre parêntesis e separados por vírgulas os componentes deste valor. Cada componente é um valor, descrito diretamente ou através de expressões envolvendo operadores. Nos exemplos de 1 a 7, todos eles usam constantes. O exemplo 7 apresenta um dos valores descrito por uma expressão. No exemplo 3, um dos termos é uma outra tupla. Qualquer termo pode ser uma tupla. Um valor pode também ser paramétrico, como no exemplo 8, onde temos uma tupla de 3 termos, todos eles paramétricos, mas o terceiro depende dos dois primeiros. Podemos dizer que esta tupla descreve todas as triplas onde o terceiro termo pode ser obtido pela soma dos dois primeiros.

Vejamos agora alguns exemplos de uso de tuplas na descrição de valores.

Exemplo 1: Desejamos definir uma função, para que dados 2 valores, seja produzido uma tripla onde os dois primeiros termos são idênticos aos elementos fornecidos, em ordem inversa, e o terceiro termo seja igual à soma dos dois.

```
def triplaS(a,b) : return (b,a,a+b)
```

Exemplo 2: Vamos definir uma função que produza o quociente e o resto da divisão inteira de dois números.

```
def divInt(a,b) : return (a//b,a%b)
```

Exemplo 3: Voltemos à definição das raízes de uma equação do 2^{o}. grau. Vimos anteriormente que como eram duas, precisávamos definir também duas funções. Agora podemos agrupá-las em uma única definição:

```
def re2g(a,b,c) :
  def x1() : return ((-b)+e())/duploA()
  def x2() : return ((-b)-e())/duploA()
  def e() : return sqrt(b**2-4*a*c)
  def duploA() : return 2*a
  return (x1(),x2())
```

Exemplo 4: Voltemos aos movimentos do cavalo no jogo de xadrez. Vamos definir uma função que produza a nova posição, usando o primeiro movimento válido segundo o que se discutiu na Seção 9.

```
def qPos(x,y) :
  def f(w): return w<=8
  def g(w): return w>=1
  if f(x+2) and f(y+1) : return (x+2,y+1)
  elif f(x+1) and f(y+2) : return (x+1,y+2)
  elif g(x-1) and f(y+2) : return (x-1,y+2)
  elif g(x-2) and f(y+1) : return (x-2,y+1)
  elif g(x-2) and g(y-1) : return (x-2,y-1)
  elif g(x-1) and f(y-2) : return (x-1,y-2)
  elif f(x+1) and f(y-2) : return (x+1,y-2)
  else : (0,0)
```

Qual o valor de qPos(1,9)? O que há de errado? Reescreva qPos(x,y) de forma a contornar o problema encontrado.

10.4. SELECIONANDO TERMOS DE UMA TUPLA: Assim com precisamos compor uma tupla na descrição dos mapeamentos de uma função, também precisamos decompô-la. Quando uma função possui tuplas como parâmetros, para usar seus termos é necessário que se possa referenciá-los. Ilustramos isso utilizando a definição que soluciona o seguinte problema:

Desejamos determinar a distância entre dois pontos no plano.

```
def dist(p1,p2):
  (x1,y1) = p1
  (x2,y2) = p2
  def dx(): return x1-x2
  def dy(): return y1-y2
  return sqrt(dx()**2+dy()**2)
```

Observe a definição da tupla (x1,y1). Por análise da definição, sabemos que o tipo de p1 e p2 é tupla de 2 termos. Quando submetemos a avaliação de uma instância, o sistema “casa” p1 com um par de valores, faz o mesmo com p2 e a partir daí pode determinar o valor de cada termo de nossa tupla (x1, y1).

```
>>> dist((0,0),(5.0,5.0))
 7.0710678118654755
```

11. Validação de Dados

11.1. INTRODUÇÃO: Como sabemos, toda função tem um domínio e um contradomínio. Em Python e na maioria das linguagens de programação, quando tentamos avaliar uma instância de uma definição, usando valores fora desse domínio, podemos receber como resposta mensagens nem sempre esclarecedoras. Quando se trata do usuário de nosso programa, esta situação se mostra mais indesejável ainda. Aqueles que constroem a definição podem discernir com mais facilidade a natureza dos problemas que ocorrem durante o seu desenvolvimento. O mesmo não se pode esperar de alguém que não tem conhecimento dos elementos internos de nossas definições.

Tipicamente os erros serão de duas naturezas. Pode ser que o tipo da instância esteja correto mas nossa definição use alguma função primitiva que não se aplica ao valor da instância. Por exemplo, se temos a definição:

```
def f(x,y,z): return (x//y + z)
```

e se a submetemos a uma instância onde ***y = 0***, teremos como resultado algo da seguinte natureza:

```
>>> f(5,0,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File
"/media/psf/Dropbox/IComp/disciplinas/ic/LivrosTutoriais
Etc/PythonCode/s11.py", line 3, in f
    def f(x,y,z): return (x//y + z)
ZeroDivisionError: integer division or modulo by zero
>>>
```

Um outro tipo de problema ocorre quando o tipo de algum parâmetro da nossa instância não casa com o tipo esperado pela linguagem. Por exemplo, se usamos um valor do tipo ***Float*** para ***y***, algumas linguagens indicarão erro, já o Python tem o seguinte comportamento:

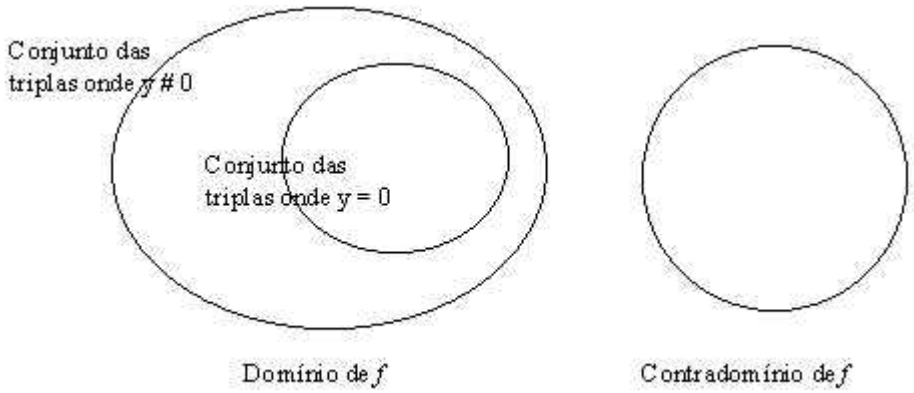
```
>>> f(5,2.0,3)
5.0
>>>
```

Nesta seção trataremos do primeiro caso, deixando o segundo para outra oportunidade.

11.2. CARACTERIZANDO A SITUAÇÃO: Voltemos à nossa definição:

```
def f(x,y,z): return (x//y + z)
```

Neste caso, o universo será formado por todos os possíveis ternos de valores onde os dois primeiros são do tipo ***Integer*** e o último, do tipo obtido pela união dos tipos ***Float*** e ***Integer.*** Chamemo-lo de ***T***. Entretanto, o domínio de nossa função não é exatamente o conjunto de constantes de ***T***, posto que a função não está definida para as constantes de ***T*** cujo segundo elemento é nulo.



Desejamos construir uma função que seja uma extensão da função original e ao mesmo tempo lhe seja uma "casca de proteção" contra as violações de domínio. Precisamos escolher um contradomínio da função estendida (***fx***). Um candidato natural é o contradomínio da função original (***f***).

E a imagem de nossa função, o que podemos dizer dela? Será que ela é igual ao contradomínio? Ou será que temos um subconjunto para o qual não exista um mapeamento?

11.3. CASO GERAL (IMAGEM IDÊNTICA AO CONTRADOMÍNIO): No caso geral, podemos encontrar duas situações: ou o contradomínio é idêntico à imagem, ou a determinação dos elementos que não pertencem à imagem pode não ser fácil. Com isso não dispomos de valores no contradomínio que possam ser utilizados para mapearmos os valores que violam o domínio de ***f***. Neste caso podemos estender o contradomínio (CD) de tal modo que o novo escolhido incorpore um valor que será a imagem da extensão de ***f***.

Uma solução geral bastante conveniente é construir um novo contradomínio (NCD) para **fx** (extensão de **f**), formado por pares onde o primeiro elemento é do tipo boolean e o segundo do contradomínio de **f**. Temos então a seguinte estrutura:

NCD(fx) = (boolean, CD(f))

Assim, para os valores de x dentro do domínio teremos:

$$fx(x) = (\text{True}, fx)$$

Para os valores de ***x*** fora do domínio teremos:

$$fx(x) = (\text{False}, k)$$

onde ***k*** é um valor qualquer pertencente ao contradomínio de ***f***.

Para o nosso exemplo inicial teríamos então:

```
def fx(x,y,z):
  if not valido(x,y,z): return (False,0)
  else: return (True,f(x,y,z))

def valido(x,y,z):
  return (pert(x,1,8) and pert(y,1,8) and pert(z,1,8))
```

Criamos assim uma casca protetora para ***f.***

11.4. FUNÇÕES COM VÁRIOS PARÂMETROS: Quando uma função possui vários parâmetros, pode ocorrer que mais de um deles deem origem à questão que aqui levantamos. Quando isso ocorre, pode ser relevante caracterizar a situação apropriadamente. Neste caso podemos usar um conjunto de constantes mais variado do que as booleanas, permitindo que possamos associar com cada erro uma constante diferente.

Podemos tomar como exemplo o problema do movimento dos cavalos no jogo de xadrez, especificamente a solução genérica que produzimos com a função

mov(m,x,y)

onde ***m*** é o número do movimento, ***x*** a linha atual e ***y*** a coluna atual.

Os três parâmetros são válidos apenas para o intervalo [1,8]. Portanto ***mov*** não está definida para os valores pertencentes ao subconjunto do universo formado por todas as triplas onde pelo menos um dos elementos não pertence ao intervalo [1,8]. Por outro lado, o contradomínio é conjunto booleano e portanto só possui 2 constantes, e ambas já estão comprometidas. Se quisermos distinguir os 3 tipos de violações do domínio (movimento inválido, posição inválida, ambos inválidos) precisaremos usar um conjunto com pelo menos 4 constantes.

Vejamos a definição a seguir:

```
def movx(m,x,y):
  if not validom(m):
    if not validop(x,y): return (3,False)
    else: return (1,False)
  else:
    if not validop(x,y): return (2,False)
    else: return (0,mov(m,x,y))

def validom(m): return pert(m,1,8)
def validop(x,y): return (pert(x,1,8) and pert(y,1,8))
```

11.5. CASO PARTICULAR (IMAGEM DIFERENTE DO CONTRADOMÍNIO): Suponha que existe pelo menos um elemento ***k*** que não pertence à imagem, ou seja, a imagem está contida no contradomínio. Podemos construir uma extensão de ***f*** de tal forma que os elementos que não pertençam ao domínio sejam mapeados neste ***k*** e os demais sejam mapeados diretamente pela ***f*** .

Podemos, portanto reescrever a nossa definição, da seguinte forma:

```
def fx(x,y,z):
  if not valido(x,y,z): return k
  else: return f(x,y,z)
```

Quando existe tal ***k***, nosso problema está resolvido. Basta que o usuário saiba que, quando a avaliação resultar em ***k***, significa que a função não se aplica para aquela instância. Infelizmente, para esse caso, o ***k*** não existe (prove!).

Voltemos ao movimento do cavalo. Nesse caso, especificamente, porque o contradomínio original é o conjunto booleano, poderíamos ter tomado outro caminho. Poderíamos usar números negativos para indicar os 3 tipos de violação do domínio, o ***0*** para representar ***False*** e o ***1*** para representar ***True***, eliminando com isso a necessidade de termos um novo domínio formado por pares. Vejamos como fica esta definição:

```
def movx2(m,x,y):
  if not validom(m):
    if not validop(x,y): return (-3)
    else: return (-1)
  else:
    if not validop(x,y): return (-2)
    elif mov(m,x,y): return (1)
    else: return (0)
```

11.6. UM EXEMPLO - RAIZES DE UMA EQUAÇÃO DO 2O. GRAU: Voltemos ao problema de determinar as raízes de uma equação do segundo grau. Já sabemos que elas são duas e que podemos fazer uma única função para descrevê-las, usando tuplas.

Sabemos ainda que o universo definido pelos tipos dos 3 parâmetros (a,b,c), é maior que o domínio da função. Ou seja, a função não está definida para instâncias de a, b e c, onde se verifica a seguinte desigualdade:

```
(b**2)-(4*a*c) < 0
```

Precisamos, portanto de uma função estendida:

```
def re2gx(a,b,c):
  def delta(): return((b**2)-(4*a*c))
  def x1(): return ((-b)+sqrt(delta()))/(2*a)
  def x2(): return ((-b)-sqrt(delta()))/(2*a)
  if delta()<0: return(False,(0,0))
  else: return(True,(x1(),x2()))
```

12. Listas

12.1. INTRODUÇÃO: No mundo de nossos problemas, a maioria dos objetos de nosso interesse estão agrupados, dando origem a um outro objeto, o agrupamento. Frequentemente estamos interessados em manipular esses agrupamentos para extrair informações, definir novos objetos ou avaliar propriedades.

Tratamos anteriormente de tuplas, que são agrupamentos de tamanho predefinido e heterogêneo. Agora estamos interessados em explorar um outro tipo de agregação, as listas. Esse novo tipo caracteriza-se por agregar quantidades variáveis de elementos usualmente de um mesmo tipo.

Vivemos cercados de listas. Elas estão em qualquer lugar onde precisamos registrar e processar dados. Vejamos alguns exemplos:

1. Lista de números pares;
2. Lista dos livros lidos por uma pessoa;
3. Lista dos amigos que aniversariam em um dado mês;
4. Lista dos presidentes corruptos;
5. Lista dos vereadores decentes;
6. Lista das farmácias enganadoras;
7. Lista das disciplinas que já cursei;
8. Lista dos lugares que visitei;
9. Lista dos números feios;
10. Lista dos números primos;
11. Lista das posições para as quais um cavalo pode se deslocar;
12. Lista das palavras de um texto;
13. Lista dos bugs provocados pelo Windows;
14. Lista dos prêmios Nobel ganhos pelo Bertrand Russel.

Destas, algumas são vazias, outras são finitas e algumas infinitas.

12.2. CONCEITOS BÁSICOS: Uma lista é uma sequência de zero ou mais elementos de um mesmo tipo.

Entende-se por sequência uma quantidade qualquer de itens dispostos linearmente.

Podemos representar uma lista pela enumeração dos seus elementos, separados por vírgulas e cercados por colchetes.

$$[\ e1,\ e2,\ \ldots,\ e_n\ ]$$

Por exemplo:

```
1. [ ]
2. [1,3,5,7,9]
3. ['a','e','i','o','u']
4. [(22,4,1500),(7,9,1822),(31,3,1964)]
5. [[1,2,5,10],[1,11],[1,2,3,4,6,12],[1,13],[1,2,7,14],
   [1,3,5,15]]
```

É importante ressaltar que em uma lista podemos falar do primeiro elemento, do quinto, ou do último. Ou seja, há uma correspondência direta entre os números naturais e a posição dos elementos de uma lista.

Este último fato nos lembra de um equívoco frequente, que queremos esclarecer de saída. A ordem que se adota em listas, por ser baseada nos números naturais, começa do zero. Ou seja, o primeiro elemento de uma lista tem o número de ordem igual a zero.

Por exemplo, a primeira lista apresentada é vazia. Na lista do item 4 acima o elemento de ordem 1 é a tupla (7,9,1822) e na lista do item 5, o elemento de ordem zero (0) é a lista [1,2,5,10].

Um elemento de uma lista é indicado por um índice, um número natural indicado entre colchetes ao lado da lista – mais sobre o assunto em breve. No caso dos exemplos mencionados no parágrafo anterior:

```
>>> [(22,4,1500),(7,9,1822),(31,3,1964)][1]
(7, 9, 1822)
>>>      [[1,2,5,10],[1,11],[1,2,3,4,6,12],[1,13],[1,2,7,14],
[1,3,5,15]][0]
[1, 2, 5, 10]
```

Quanto ao tipo, podemos dizer que a segunda lista é uma lista de números, a terceira uma lista de caracteres, a quarta é uma lista de triplas de números e a quinta é uma lista de listas de números. Qual será o tipo da primeira lista?

Uma lista vazia é de natureza polimórfica, isto é, seu tipo depende do contexto em que seja utilizada, como veremos em momento oportuno.

12.3. FORMAS ALTERNATIVAS PARA DEFINIÇÃO DE LISTAS: Além da forma básica, acima apresentada, também conhecida como enumeração, onde explicitamos todos os elementos, dispomos ainda das seguintes maneiras: intervalo, progressão aritmética e por compreensão, onde as duas primeiras são produzidas pela função `range`, conforme mostrado a seguir.

Intervalo

De uma forma geral, podemos definir uma lista explicitando os limites inferior e superior de um conjunto conhecido, onde existe uma relação de ordem entre os elementos, no Python esse conjunto pode ser referenciado assim:

range(<limite inferior>**,**<limite superior>**)**

Quando omitido o limite inferior, zero é assumido. No intervalo definido por range, o limite inferior é fechado e o superior é aberto, observe os exemplos a seguir:

```
>>> range(1,6)
range(1, 6)
>>> list(range(1,6))
[1, 2, 3, 4, 5]
>>> list(range(-2,2))
[-2, -1, 0, 1]
>>> list(range(10,5))
[]
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(-5,5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

Progressão Aritmética

Podemos definir qualquer progressão aritmética por uma lista utilizando a seguinte notação:

range(<1o. termo>,<limite superior>,<razão>**)**

Quando a razão é omitida, o valor 1 é assumido. Observe os exemplos a seguir:

```
 >>> list(range(1,7,1))
[1, 2, 3, 4, 5, 6]
>>> list(range(-5,6,7))
[-5, 2]
>>> list(range(-5,17,7))
[-5, 2, 9, 16]
>>> list(range(6,0,-1))
[6, 5, 4, 3, 2, 1]
>>> list(range(6,0,-2))
[6, 4, 2]
>>>
```

12.4. OPERAÇÕES BÁSICAS: As listas, como já dissemos, são elementos da linguagem que podemos utilizar para descrever valores estruturados. Como nos demais tipos da linguagem, valores descritos por listas podem e devem ser usados na descrição de novos valores através da utilização de operações definidas sobre eles. A seguir são apresentadas algumas dessas operações.

len : descreve o tamanho de uma lista.

len(<lista>)

```
>>> len(range(10))
10
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> len(range(10))+len(range(10))
20
>>>
```

indexação : podemos descrever como um novo valor, cada termo de uma lista. Para tanto basta indicarmos a posição do elemento dentro da lista, considerando que o primeiro elemento tem a ordem zero.

<lista>**[**<índice1>**][**<índice2>**]...[**<índiceN>**]**

```
>>> [3,4,6,74,45][0]
3
>>> [3,4,6,74,45][4]
45
>>> range(1,21,4)[3]
13
>>> [[[1],[2,3,4]]][0][1][2]
4
>>> [3,4,6,74,45][3]
74
>>> [3,4,6,74,45][3] + [2,3][0]
76
>>> [3,4,6,74,45][0+1]
4
>>> i=3
>>> j=0
>>> [3,4,6,74,45][i+j]
74
>>> range(1,21,4)[5]
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
IndexError: range object index out of range
```

concatenação : descreve uma nova lista obtida pela concatenação de uma lista de listas.

```
<lista1> + <lista2> + ... + <listan>
```

```
>>> [1]+[1]
[1, 1]
>>> [1]+[]
[1]
>>> list(range(1,6))+list(range(1,101,9))+[9]
[1, 2, 3, 4, 5, 1, 10, 19, 28, 37, 46, 55, 64, 73, 82, 91,
100, 9]
>>>
```

inclusão : descreve uma nova lista onde o último termo é um dado elemento e os demais são os componentes de uma lista também dada.

```
<lista>.append(<elemento>)
```

```
>>> x=[1,2,3]
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>> y=[[3]]
>>> y.append([4,5,6])
>>> y
[[3], [4, 5, 6]]
```

sublistas: uma nova lista pode ser obtida a partir de “fatias” da lista original. No Python isso é possível indicando os limites inferior e superior para as referidas “fatias”, na seguinte forma:

```
<lista>[<limite inferior>:<limite superior>]
```

Quando o limite inferior ou o limite superior são omitidos, o primeiro e o último elementos da lista são, respectivamente, assumidos. Valores negativos indicam os elementos antecessores ao último da lista, como pode ser observado nos exemplos a seguir:

```
>>> list(range(1,11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(1,11))[3:7]
```

```
[4, 5, 6, 7]
>>> list(range(1,11))[1:7]
[2, 3, 4, 5, 6, 7]
>>> list(range(1,11))[:7]
[1, 2, 3, 4, 5, 6, 7]
>>> list(range(1,11))[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(1,11))[:-1]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1,11))[:-2]
[1, 2, 3, 4, 5, 6, 7, 8]
>>> list(range(1,11))[-4:-2]
[7, 8]
>>> list(range(1,11))[:999]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>>
```

12.5. DEFINIÇÃO POR COMPREENSÃO: Uma lista pode ser descrita através da enumeração de seus elementos, como já vimos através de vários exemplos. Esta forma é também denominada de definição por extensão, visto que todos os componentes precisam ser explicitados.

Podemos também descrever listas através das condições que um elemento deve satisfazer para pertencer a ela. Em outras palavras, queremos descrever uma lista através de uma intenção. Esta forma é análoga à que já conhecemos para descrição de conjuntos. Por exemplo, é usual escrever a notação abaixo para descrever o conjunto formado pelo quadrado dos números naturais menores que 10.

P = { quadrado de x | x é natural e é menor que 10}

ou ainda mais formalmente,

P = { x^2 | x pertence a N e x<10 }

Podemos observar, no lado direito da definição, que ela é formada por duas partes. A primeira é uma expressão que descreve os elementos, usando para isso termos variáveis que satisfazem condições de pertinência estabelecidas pela segunda parte.

expressão	x^2
variável	x
pertinência	x pertence a N e x<10

Em nosso caso, não estamos interessados em descrever conjuntos e sim listas. E isso tem algumas implicações práticas. Por exemplo, em um conjunto a

ordem dos elementos é irrelevante, para listas não. É bom lembrar ainda que em uma lista, o mesmo valor pode ocorrer varias vezes, em diferentes posições.

A sintaxe que usaremos é a seguinte:

[<expressão> **for** <variável> **in** <fonte> **if** <condição>**]**

Onde <expressão> é qualquer expressão usual em Python para definição de valores, que usualmente inclui <variável> obtida de <fonte>, que é uma estrutura sequenciada, normalmente uma lista. A última parte da estrutura (if <condição>) é opcional e condiciona a inclusão de cada elemento de <fonte> à verificação de <condição>.

Por exemplo, vejamos a descrição da lista dos quadrados dos números menores que 5.

```
>>> [x**2 for x in range(5)]
[0, 1, 4, 9, 16]
>>>
```

Vejamos o exemplo a seguir, onde descrevemos uma sublista de números ímpares, tendo como origem de geração uma lista definida por uma Progressão Aritmética.

```
>>> [x for x in range(1,100,3) if x%2!=0]
[1, 7, 13, 19, 25, 31, 37, 43, 49, 55, 61, 67, 73, 79, 85,
91, 97]
>>>
```

Vejamos como usar este novo conceito na escrita de novos scripts. No quadro abaixo apresentamos a definição de três novas funções. A primeira, slpares, define uma sublista formada pelos quadrados dos elementos pares de uma lista dada. A segunda, lmenor, define uma sublista formada pelos elementos de uma dada lista, que são menores que um elemento fornecido. A terceira, pmaioresk, ilustra a geração de uma lista com o dobro ou o quadrado dos valores originais de uma lista dada, dependendo se cada elemento é ou não maior que um valor k.

```
# Dada uma lista xs, define uma sublista formada pelo
quadrado dos elementos que são pares
def slpares(xs):
  return [x**2 for x in xs if par(x)]
def par(x): return x%2==0

# Sublista de elementos menores que x em uma lista xs
def lmenor(x,xs):
  return [y for y in xs if y<x]
```

```
# Lista de dobros ou quadrados dos elementos de xs
def pmaioresk(k,xs):
  return [dobroquad(x,k) for x in xs]
def dobroquad(x,k):
  if x>k: return 2*x
  else: return x**2

>>> slpares(range(1,50,3))
[16, 100, 256, 484, 784, 1156, 1600, 2116]
>>> slpares([34,67,99,23,12,3,67,99])
[1156, 144]
>>> lmenor(45,[1,5,6,86,34,76,12,34,86,99])
[1, 5, 6, 34, 12, 34]
>>> lmenor(1,[1,5,6,86,34,76,12,34,86,99])
[]
>>> pmaioresk(30,[1,5,6,86,34,76,12,34,86,99])
[1, 25, 36, 172, 68, 152, 144, 68, 172, 198]
>>>
```

Quando mais de um gerador (<fonte>) é utilizado, devemos levar em conta que para cada elemento do gerador mais a esquerda serão gerados todos os elementos dos geradores subsequentes. Vejamos o exemplo a seguir onde descreve-se uma lista de pontos do plano cartesiano, localizados no primeiro quadrante e delimitado pelas ordenadas 3 e 5.

```
# Determinar a lista dos pontos do plano dentro da regiao
definida pela origem, a coordenada (eixo y) 5 e a abscissa
(eixo x) 3.
def pontos():
  return [(x,y) for x in range(4) for y in range(6)]

>>> pontos()
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 0), (1,
1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 0), (2, 1), (2, 2),
(2, 3), (2, 4), (2, 5), (3, 0), (3, 1), (3, 2), (3, 3), (3,
4), (3, 5)]
>>>
```

Entre dois geradores podemos usar predicativos, como se observa no exemplo a seguir.

```
# Determinar a lista dos pontos do plano dentro da regiao
definida pela origem, a coordenada (eixo y) 5 e a abscissa
(eixo x) 3, considerando apenas x impares e y pares.
def pontos1():
```

```
  return [(x,y) for x in range(4) if (x%2!=0) for y in
range(6) if (y%2==0)]

>>> pontos1()
[(1, 0), (1, 2), (1, 4), (3, 0), (3, 2), (3, 4)]
>>>
```

12.6. DEFINIÇÃO POR COMPREENSÃO - EXPLORANDO DETALHES: Vamos explorar um problema onde exista mais riqueza de detalhes quanto ao uso de predicativos.

Determinar os pares de valores, onde:

- o primeiro é múltiplo do segundo;
- o primeiro é dado pelos elementos impares de uma P.A de primeiro termo igual a 1, a razão igual 3 e o ultimo termo menor ou igual a 100;
- o segundo termo é dado pelos elementos de uma P.A de primeiro termo igual a 1, a razão igual 4 e o ultimo termo menor ou igual a 50;
- um dos dois é diferente da unidade;
- os dois termos são diferentes.

```
'''
A função paresE, traduz literalmente o enunciado.
As P.A.'s são realizadas diretamente pelo mecanismo embutido
na linguagem.
Os predicativos foram inseridos na posição em que suas
variáveis já estão instanciadas.
'''
def paresE1():
  return [(x,y)
          for x in range(1,100,3)
          if (x%2!=0)
          for y in range(1,50,4)
          if (x!=1 or y!=1) and (x!=y) and x%y==0]

>>> paresE1()
[(7, 1), (13, 1), (19, 1), (25, 1), (25, 5), (31, 1), (37,
1), (43, 1), (49, 1), (55, 1), (55, 5), (61, 1), (67, 1),
(73, 1), (79, 1), (85, 1), (85, 5), (85, 17), (91, 1), (91,
13), (97, 1)]
>>>
```

Vejamos algumas observações sobre a solução acima apresentada:

- a verificação se o valor de x é impar `(x%2!=0)` poderia ser colocado em qualquer lugar mais a frente, entretanto o desempenho cairia, visto que iríamos produzir valores desnecessários para "y";

- a expressão `(x!=1 or y!=1)` é desnecessária visto que só será falsa quando ambos, x e y, forem iguais a 1, mas nesse caso eles serão iguais e portanto falsificariam a expressão a seguir `(x!=y)`;
- podemos reescrever a expressão `range(1,100,3)` **d**e tal maneira que gere apenas valores impares e assim descartar a expressão `(x%2!=0)`. Para tanto basta mudar a P.A. para `range(1,100,6)`**;**
- pode-se observar que os valores de **y** que interessam são sempre menores que os de **x** (já que **y** é divisor de **x**). Portanto, a segunda P.A. poderia ser substituída por `range(1,x,4)`. Acontece que agora poderemos geram valores para **y** maiores que 50 e isto não interessa. O que fazer? Que tal substituí-la por:

```
range(1,(x//50+x%50),4)
```

Eis então uma nova versão para nossa função:

```
def paresE2():
  return [(x,y)
          for x in range(1,100,6)
          for y in range(1,(x//50+x%50),4)
          if (x!=y) and x%y==0]
```

Podemos agora refletir sobre uma possível generalização para a nossa função, considerando-se duas listas quaisquer. Neste caso, o esforço realizado para melhorar o desempenho seria em vão porque não conhecemos a priori a natureza das duas listas. Nossa função poderia ser:

```
def paresE3(xs,ys):
  return [(x,y)
          for x in xs
          if (x%2!=0)
          for y in ys
          if (x!=y) and x%y==0]
```

Apenas a expressão `(x!=1 or y!=1)` poderia ser eliminada. O objetivo de nossa versão original poderia ser obtido pelo uso da nova função aplicada às listas específicas. Conforme se observa a seguir:

```
>>> paresE3(range(1,100,3),range(1,50,4))
[(7, 1), (13, 1), (19, 1), (25, 1), (25, 5), (31, 1), (37,
1), (43, 1), (49, 1), (55, 1), (55, 5), (61, 1), (67, 1),
(73, 1), (79, 1), (85, 1), (85, 5), (85, 17), (91, 1), (91,
13), (97, 1)]
>>>
```

12.7. EXERCÍCIO: OPERAÇÕES PARA DETERMINAR SUBLISTAS

Em algumas linguagens existem funções predefinidas para descrevermos sublistas de uma lista dada. Essas funções podem ser criadas com o que já apresentamos até aqui e o exercício de defini-las pode ajudar na prática do reuso e contribuir bastante para a clareza de um programa:

```
# Lista com os k primeiros elementos de uma lista xs
def take(k,xs): ?

# Lista com os elementos de xs seguintes aos k primeiros
def drop(k,xs): ?

# Primeiro elemento de uma lista xs
def head(xs): ?

# Sublista similar a xs mas sem o primeiro elemento
def tail(xs): ?

# Ultimo elemento de uma lista xs
def last(xs): ?

# Sublista similar a xs mas sem o ultimo elemento
def init(xs): ?
```

13. Resolvendo Problemas com Listas

13.1. DETERMINANDO O MAIOR ELEMENTO:

Dada uma lista, determine o seu maior elemento.

Comecemos por definir, usando a linguagem de conjuntos, quem é este elemento. Dizemos que k é o maior elemento de um conjunto C, se e somente se, o subconjunto de C formado por todos os elementos maiores que k é vazio.

Em “linguagem de listas” isto equivale a dizer que se C é uma lista, a sublista de C formada pelos caras de C maiores que k é vazia. O “cabeça” dessa lista é o máximo dela, como no exemplo a seguir:

```
# Vamos construir uma função para determinar, em uma lista, a
sublista dos elementos maiores que um dado x
def maiores(x,xs):
  return [ y for y in xs, y > x]

# Em listas, podemos ter elementos repetidos, e em particular
podemos ter vários exemplares do maior elemento.
# Chamemos esses caras de os "maiorais da lista". Vamos
construir uma função para descrevê-los.
def maiorais(xs):
  return [ k for k in xs, if maiores(k,xs)==[] ]

# Como eles são todos idênticos podemos tomar o primeiro
deles como solução de nosso problema.
def maximo(x)
  return head(maiorais(xs))
```

Apesar do Python ter uma função pré-definida com esse propósito, sua definição é um exercício interessante e útil para solucionar outros problemas.

13.2. LISTAS NÃO DECRESCENTES: Como aquecimento a problemas que envolvam a ordenação de listas, vamos considerar listas de números e a noção usual de ordem não decrescente. Nosso problema é:

Dada uma lista, verifique se ela é não decrescente.

Antes de programar, vamos resgatar a definição de sequências não decrescentes.

Def : Uma seqüência S está em ordem não decrescente se e somente se qualquer um de seus elementos é menor ou igual aos seus sucessores. Em outras

palavras, podemos dizer que a coleção de elementos de S que são maiores que seus sucessores é vazia.

```
# lista dos maiores que os sucessores
def lms1(xs):
  return [ xs[i]  for i in range(0,(len(xs)-1))
                  for j in range((i+1),len(xs))
                  if xs[i] > xs[j] ]

# a sequência está ordenada se a lista dos elementos que são
maiores que algum sucessor é vazia
def ord1(xs):
  return lms1(xs)==[]

>>> ord1([1,2,3,19,32])
True
>>> ord1([1,2,3,21,19,32])
False
>>>
```

13.3. DISCUTINDO EFICIÊNCIA: Em geral os motivos que afetam o desempenho o desempenho de uma solução são de duas naturezas: exploração inadequada das propriedades do problema e escolha inadequada dos mecanismos da linguagem. A seguir fazemos uma pequena exploração desses dois aspectos.

13.3.1. EXPLORANDO PROPRIEDADES DO PROBLEMA - Analisando a nossa definição anterior constatamos que ela diz mais do que precisamos. Ela avalia cada elemento com respeito a todos sucessores. Na verdade, nossa definição pode ser melhorada. Basta saber que cada elemento tem que ser menor ou igual ao seu sucessor imediato. Vamos reescrever a nossa definição:

Def : Uma sequência S está em ordem não decrescente se e somente se qualquer um de seus elementos é menor ou igual ao seu **sucessor imediato**. Em outras palavras, podemos dizer que a coleção de elementos de S que são maiores que seus **sucessores imediatos** é vazia.

Vejamos então a implementação em Python e sua aplicação às mesmas instâncias do problema:

```
# lista dos elementos maiores que o sucessor imediato
def lms2(xs):
  return [ xs[i]  for i in range(0,(len(xs)-1))
                  if xs[i] > xs[i+1] ]
def ord2(xs):
  return lms2(xs)==[]

>>> ord2([1,2,3,19,32])
```

```
True
>>> ord2([1,2,3,21,19,32])
False
>>>
```

13.3.2. EXPLORANDO OS MECANISMOS DA LINGUAGEM - Uma outra investida que podemos fazer é com respeito ao uso adequado dos mecanismos da linguagem. As soluções acima apresentadas processam as listas através de índices, ou seja, fazem um acesso aleatório aos elementos da lista. Sabendo-se que o acesso sequencial possui realização mais eficiente, poderíamos usar essa característica em nossa solução:

```
# Para manusear os pares de adjacentes, ao invés de usar índices, usemos a função zip aplicada a um par de listas construídas com base em xs
# a)lista formada pelos elementos de xs, exceto o último, que pode ser obtida com a função init;
# b)lista formada pelos elementos de xs, exceto o primeiro, que pode ser obtida com a função tail.
# Com isso obtemos uma lista formada pelos pares adjacentes.
def adjacentes(xs):
  return zip(init(xs),tail(xs))

#   A nova função para definir lista dos maiores que os sucessores. Agora trabalharemos com os pares
def lms3(ps):
  return [ (x,y) for (x,y) in ps if x>y ]

# A nova versão de 'ord'
def ord3(xs):
  return lms3(adjacentes(xs))==[]

>>> list(adjacentes([1,2,3,4,5,6,7,8,9,10]))
[(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9), (9, 10)]
>>> ord3([1,2,3,4,5,6,7,8,9,10])
True
>>> ord3([1,2,3,4,-5,6,7,8,9,10])
False
>>>
```

14. O Paradigma Aplicativo

14.1. INTRODUÇÃO: A descrição de funções, como acontece com outras formas de representação de conhecimento, admite vários estilos. Dependendo do problema que estamos tratando, um estilo pode ser mais conveniente que outro. Podemos dizer que influi muito na escolha, o quanto desejamos ou necessitamos, falar sobre como computar uma solução. A descrição de soluções tem um espectro de operacionalidade que vai do declarativo até o procedural, em outras palavras, do **que** desejamos computar ao **como** queremos computar.

Uma situação que ocorre com frequência na descrição de funções é a necessidade de aplicar uma função, de forma cumulativa, à uma coleção de elementos. Em outras palavras, desejamos generalizar uma operação para que seu uso se estenda a todos os elementos de uma lista. Chamemos este estilo de paradigma aplicativo.

Por exemplo, suponha que desejamos obter a soma de todos os elementos de uma lista. A operação ***adição* (+)** segundo sabemos, é de natureza binária, ou seja, opera sobre dois elementos produzindo um terceiro. Para operar sobre todos os elementos de uma lista de forma a obter a soma total, podemos operá-los dois a dois, obtendo com isso resultados intermediários, que por sua vez poderão ser novamente operados e assim sucessivamente até que se obtenha o resultado final. Observemos o processo para uma instância:

Obter a soma dos elementos da lista [5, 9, 3, 8, 15, 16]

expressão	nova expressão	redução
+ [5, 9, 3, 8, 15, 16]	+ [14, 3, 8, 15, 16]	5 + 9 = 14
+ [14, 3, 8, 15, 16]	+ [14, 11, 15, 16]	3 + 8 = 11
+ [14, 11, 15, 16]	+ [14, 11, 31]	15 + 16 = 31
+ [14, 11, 31]	+ [25, 31]	14 + 11 = 25
+ [25, 31]	+ [56]	25 + 31 = 56
+ [56]	56	

As reduções poderiam ser aplicadas em qualquer ordem, para o caso da adição, visto que a operação é comutativa. Assim sendo, podemos estabelecer uma ordem, por exemplo, da esquerda para a direita, usando em cada nova redução o elemento resultante da redução anterior.

Expressão	nova expressão	Redução
+ [5, 9, 3, 8, 15, 16]	**+** [14, 3, 8, 15, 16]	5 + 9 = 14
+ [14, 3, 8, 15, 16]	**+** [17, 8, 15, 16]	14 + 3 = 17
+ [17, 8, 15, 16]	**+** [25, 15, 16]	17 + 8 = 25
+ [25, 15, 16]	**+** [40, 16]	25 + 15 = 40
+ [40, 16]	**+** [56]	40 + 16 = 56
+ [56]	56	

Em Python existe um operador que permite a descrição de computações desta natureza. Este operador denomina-se ***reduce***. Os dois primeiros elementos da lista são tomados como ponto de partida para as reduções. Eis a sintaxe:

reduce (<operação>,<lista>**)**

Observe que no Python3, para usar esse operador, é necessário “importa-lo” através da declaração

```
from functools import reduce
```

Também é interessante saber que é possível definir um valor especial, um “inicializador” que para gerar o primeiro resultado de <operação>, é utilizado como um dos operandos sendo o outro o primeiro elemento de <lista>, ou como “default” no caso de uma lista vazia. O formato é o seguinte:

reduce (<operação>,<lista>,<inicializador>**)**

A ordem estabelecida para as reduções é semelhante à ilustração acima, ou seja, caminha-se da esquerda para direita, usando o resultado da redução anterior para a nova redução.

Vejamos então a definição do operador ***sum*** (disponível no Python) cujo objetivo é a descrição da soma dos elementos de uma lista.

```
def mais(a,b):
  return a+b
def sum(xs):
  return reduce(mais,xs,0)
```

OBS:

1. A função `mais` é apenas uma maneira simples de nos a soma de dois números.
2. O valor especial é o zero, visto que não desejamos que o valor especial modifique o resultado da soma de todos os elementos. Entretanto, ele tem um papel muito especial quando a lista for vazia.
3. Em exemplos futuros veremos outros usos para o valor especial.

14.2. OPERAÇÕES BÁSICAS: Assim como a somatória, existem outras operações denominadas básicas, por serem de grande utilidade, que podem ser obtidas pelas seguintes equações usando ***reduce.***

```
# produto - valor especial = 1
def product(xs):
  return reduce(vezes,xs,1)
def vezes(a,b): return a*b

# conjunção - valor especial = True
def and_all(xs):
  return reduce(e,xs,True)
def e(a,b): return a and b

# disjunção - valor especial = False
def or_all(xs):
  return reduce(ou,xs,False)
def ou(a,b): return a or b
```

Exemplo 01: Podemos usar a função ***product*** para descrever o fatorial de um número. Para tanto basta lembrar a definição usual de fatorial: "*O fatorial de um número natural n>0 é igual ao produto de todos os número de 1 até n.*"

```
# fatorial x = produto da lista [1,2,3,...,n]
def fat(n): return product(range(1,n+1))

>>> fat(5)
120
>>> fat(0)
1
>>> fat(1)
1
```

Exemplo 02: Podemos usar a disjunção (**or_all**) generalizada para definir uma função que avalia se em uma dada lista de números pelo menos um deles é impar.

```
 # pelo menos um impar
def umImpar(xs): return or_all([x%2!=0 for x in xs])

>>> umImpar([2,4,6,8,10,11,12])
True
>>> umImpar([2,4,6,8,10,110,12])
False
>>> umImpar([2,4,6,8,10,110,121])
True
>>> umImpar([])
False
```

14.3. O MENOR ELEMENTO DE UMA LISTA: Estamos interessados em obter uma função que associe uma lista xs com o elemento de xs que seja o menor de todos.

Anteriormente já apresentamos uma versão para a função que descreve o maior elemento de uma lista, que é bastante similar a esta. Na oportunidade exploramos uma propriedade que o elemento ***maior*** de todos deve satisfazer. No caso do menor elemento, podemos explorar uma propriedade análoga: Em uma lista xs, dizemos que k é o menor elemento de xs, se e somente se a sublista de xs formada por elementos menores que k é vazia.

```
# ´menores´ descreve os elementos menores que um dado x em
uma lista xs
def menores(x,xs):
  return [ y for y in xs if y < x]

# 'minimo' descreve a sublista de xs dos elementos que não
possuem menores que eles em xs
def minimos(xs):
  return [ k for k in xs if menores(k,xs)==[] ]

# Como eles são todos idênticos podemos tomar o primeiro
deles como solução de nosso problema.
def menorL0(x):
  return head(minimos(xs))
```

Vamos explorar agora o problema a partir da generalização da operação **menor.** Em sua forma básica, a função **menor** associa dois números quaisquer com o menor entre eles. Precisamos identificar um elemento que não interfira no resultado para fazer o papel de ***valor especial***. Para a operação **menor** podemos observar que este papel pode ser desempenhado por qualquer um dos elementos

da lista, visto que o menor entre dois valores idênticos é o próprio valor. Como pode ser qualquer um, podemos escolher o primeiro elemento de xs (head).

```
# menor de dois
def menor(x,y):
  if x < y : return x
  else : return y

# menor da lista
def menorL1(xs):
  return reduce(menor,xs,head(xs))

>>> menorL1([5,5,4,4,4,6,6,6,3,3,3,11,1,0])
0
>>> menorL1([5,5,4,4,4,6,6,6,3,3,3,11,-1,0])
-1
>>> menorL1([5])
5
>>> menorL1([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File
"/Users/alberto/Dropbox/IComp/disciplinas/ic/LivrosTutoriaisE
tc/PythonCode/s14.py", line 50, in menorL1
    return reduce(menor,xs,head(xs))
  File
"/Users/alberto/Dropbox/IComp/disciplinas/ic/LivrosTutoriaisE
tc/PythonCode/s12.py", line 61, in head
    def head(xs): return xs[0]
IndexError: list index out of range
>>>
```

Podemos observar aqui que a função **menorL0** é parcial pois não se aplica a lista vazia.

14.4. INSERÇÃO ORDENADA E ORDENAÇÃO DE UMA LISTA: A ordenação de dados é uma das operações mais realizadas em computação. Diariamente, em todos os computadores do mundo, faz-se uso intensivo dela. Este assunto é muito especial e por isso mesmo profundamente estudado. Cabe-nos aqui fazer uma breve passagem pelo assunto, sem contudo nos aprofundarmos nas questões de eficiência, que é central no seu estudo.

Será que podemos usar o conceito de generalização de uma operação para descrever a ordenação de uma lista? Que operação seria essa?

14.4.1. INSERÇÃO EM LISTA ORDENADA - Vamos começar discutindo uma outra questão mais simples: dada uma lista, com seus elementos já dispostos em ordem não decrescente, como descrever uma lista na mesma ordem, acrescida de um elemento também fornecido?

Podemos observar que se a lista xs está em ordem não decrescente, com respeito ao elemento x (dado), podemos descrevê-la através de dois segmentos:

xs = <menores que x> + <maiores ou iguais a x>

Para acrescentar x a xs, basta concatenar x entre os dois segmentos, obtendo a nova lista ys, assim:

ys = <menores que x em xs> + [x] + <maiores ou iguais a x em xs>

```
# inserção ordenada
def insord(xs,x):
  def menorx(y): return y<x
  return  ( list(takewhile(menorx,xs))
          + [x]
          + list(dropwhile(menorx,xs)) )

>>> insord([],10)
[10]
>>> insord([10],20)
[10, 20]
>>> insord([10,20],30)
[10, 20, 30]
>>> insord([10,20,30],5)
[5, 10, 20, 30]
>>> insord([5,10,20,30],25)
[5, 10, 20, 25, 30]
>>>
```

14.4.2. ORDENAÇÃO - A aplicação sucessiva de **insord**, conforme ilustrada acima, nos dá uma pista para nossa generalização. Podemos pegar cada elemento da lista a ser ordenada e inseri-lo em ordem em uma outra lista que será paulatinamente construída, já ordenada. Vamos seguir o exemplo acima, onde desejamos ordenar a lista [10,20,30,5,25]:

lista parcial	novo elemento	redução
[]	10	[10]
[10]	20	[10,20]

[10,20]	30	[10,20,30]
[10,20,30]	5	[5, 10, 20, 30]
[5, 10, 20, 30]	25	[5, 10, 20, 25, 30]

O ponto de partida, neste caso, é a lista vazia. Vamos tomar então a lista vazia como o valor especial. Vejamos como fica então nossa definição para ordenação de listas.

```
# ordenação
def ordena(xs):
  return reduce(insord,xs,[])

>>> ordena([10,20,30,5,25])
[5, 10, 20, 25, 30]
>>> ordena(range(11))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> ordena(range(11,-1,-1))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>>
```

14.5. INVERSÃO DE UMA LISTA: Dada uma lista xs, desejamos descrever a lista formada pelos elementos de xs tomados em ordem inversa. Para resolver o problema precisamos inventar uma função básica passível de generalização.

Vamos começar descrevendo a função **insAntes.**

```
# Insere um elemento antes do primeiro elemento de uma dada
lista.
def insAntes(xs,x):
  ys = xs
  ys.insert(0,x)
  return ys

>>> insAntes([1,2,3,4,5],0)
[0, 1, 2, 3, 4, 5]
>>> insAntes([],10)
[10]
>>>
```

Tentemos agora a generalização. A intenção é incluir cada elemento da lista xs que desejamos inverter, antes do primeiro elemento de uma lista que iremos construindo gradativamente. O ***valor especial*** será a lista vazia. Vejamos um exemplo onde inverteremos a lista [0, 1, 2, 3, 4]

lista parcial	novo elemento	redução
[]	0	[0]
[0]	1	[1, 0]
[1, 0]	2	[2, 1, 0]
[2, 1, 0]	3	[3, 2, 1, 0]
[3, 2, 1, 0]	4	[4, 3, 2, 1, 0]

Vamos então usar o operador reduce para construir a generalização desejada:

```
# inversao
def inverte(xs):
  return reduce(insAntes,xs,[])

>>> inverte([1,2,3,4])
[4, 3, 2, 1]
>>> inverte([10,-2,3,14,171,5])
[5, 171, 14, 3, -2, 10]
>>>
```

Observe a similaridade entre a função **reverse**, pré-definida, com a que acabamos de definir.

14.6. INTERCALAÇÃO DE LISTAS: Dadas duas lista xs e ys, ambas em ordem não decrescente, desejamos descrever uma nova lista em ordem não decrescente, formada por todos os elementos das duas listas.

Um processo bastante conhecido, chamado de ***balance line***, consiste em ir transferindo para a nova lista, os elementos de uma das listas de entrada, enquanto estes forem menores que os da outra. Quando a condição não é mais satisfeita, fazemos o mesmo para a outra lista.

Por exemplo, vamos intercalar as listas [2, 4, 6, 8] e [1, 3, 5]. Vejamos o desenrolar do processo.

lista parcial	Andamento em xs	Andamento em ys	Redução
[]	[2, 4, 6, 8]	[1, 3, 5]	[1]
[1]	[2, 4, 6, 8]	[3, 5]	[1, 2]
[1, 2]	[4, 6, 8]	[3, 5]	[1, 2, 3]
[1, 2, 3]	[4, 6, 8]	[5]	[1, 2, 3, 4]
[1, 2, 3, 4]	[6, 8]	[5]	[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]	[6, 8]	[]	[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]	[8]	[]	[1, 2, 3, 4, 5, 6, 8]
[1, 2, 3, 4, 5, 6, 8]	[]	[]	

Vamos precisar aqui de uma bela engenharia para a construção da função a generalizar. Desta vez, nossa entrada é formada por duas listas e como sabemos o operador **foldl**, a cada vez, processa um elemento de uma determinada lista. Isto nos leva a ter que inventar também a lista a ser processada.

A função básica pode ser inventada a partir da descrição de um passo do processo que denominamos de ***balance line***. Em cada passo temos como entrada uma lista parcialmente construída e duas listas parcialmente processadas. Como saída teremos mais um passo de construção da lista resultante e o andamento no processamento de uma das listas. Na lista em construção devemos inserir o menor entre os cabeças (head) das duas listas. Aquela que tiver o menor cabeça dará origem a uma nova lista, da qual foi excluído o primeiro elemento, ou seja, será idêntica ao resto (tail) da lista escolhida.

```
# Descrição de um passo do Balance Line:
# Quando uma das duas listas for vazia a outras é diretamente
concatenada no final da lista em construção.
def passoBL(xyz):
  (xs,ys,zs) = xyz
  if ys==[] : return ([],[],zs+xs)
  elif xs==[] : return ([],[],zs+ys)
  elif head(xs)<=head(ys): return (tail(xs),ys,zs+[head(xs)])
  else : return (xs,tail(ys),zs+[head(ys)])

>>> passoBL(([2,4,6,8],[1,3,5],[]))
([2, 4, 6, 8], [3, 5], [1])
>>> passoBL(([2,4,6,8],[3,5],[1]))
([4, 6, 8], [3, 5], [1, 2])
>>> passoBL(([4,6,8],[3,5],[1,2]))
([4, 6, 8], [5], [1, 2, 3])
>>> passoBL(([4,6,8],[5],[1,2,3]))
```

```
([6, 8], [5], [1, 2, 3, 4])
>>> passoBL(([6,8],[5],[1,2,3,4]))
([6, 8], [], [1, 2, 3, 4, 5])
>>> passoBL(([6,8],[],[1,2,3,4,5]))
([], [], [1, 2, 3, 4, 5, 6, 8])
```

A generalização, de forma que possamos aplicar a função gradativamente, de tal forma que todo o processo se complete, requer a invenção de uma lista sobre a qual ocorra a repetição. O número de aplicações sucessivas será no máximo igual à soma dos comprimentos das listas a serem intercaladas. Podemos então definir a aplicação sobre a lista dos números de 1 até a soma dos tamanhos.

```
# Função para Intercalar duas listas ordenadas
# 1) a função passoBL é repetidamente aplicada, sobre o resultado obtido no passo anterior;
# 2) a função passoBL é binária, como exigido pelo reduce;
# 3) a aplicação sucessiva é controlada pelo comprimento da lista resultante;
# 4) o resultado final é obtido pela seleção do terceiro elemento da tripla através do indice
def baLine(xs,ys):
  return baLine3(xs,ys)[2]

def baLine3(xs,ys):
  tr = len(xs)+len(ys)+1
  return reduce(passoBL2,list(range(1,tr)),(xs,ys,[]))

# Descrição de um passo do Balance Line.
# 1) Quando uma das duas listas for vazia a outras é diretamente concatenada no final da lista em construção.
# 2) O parâmetro k é apenas para estabelecer o tipo binário exigido por reduce
def passoBL2(xyz,k):
  (xs,ys,zs) = xyz
  if ys==[] : return ([],[],zs+xs)
  elif xs==[] : return ([],[],zs+ys)
  elif          head(xs)<=head(ys)          :          return (tail(xs),ys,zs+[head(xs)])
  else : return (xs,tail(ys),zs+[head(ys)])

>>> baLine3([1,2,3],[4,5,6])
([], [], [1, 2, 3, 4, 5, 6])
>>> baLine([1,2,3],[4,5,6])
[1, 2, 3, 4, 5, 6]
>>> baLine([2,4,6,8],[1,3,5])
[1, 2, 3, 4, 5, 6, 8]
>>>
```

15. Processamento de Cadeias – primeiros passos

15.1. INTRODUÇÃO: Além de números, nosso mundo é povoado por textos. Cada vez mais se torna presente o uso de computadores para nos auxiliar na tarefa de armazenar, recuperar e processar documentos. Neste capítulo estamos interessados em fazer uma breve introdução ao uso de computadores nessas tarefas.

O ponto de partida é o tipo caractere (chr), que nos permite representar textos na memória (principal e secundária) dos computadores. Veremos também como agrupá-los para compor palavras, frases e por fim documentos.

15.2. O TIPO CHAR: O tipo char é formado por um conjunto de símbolos. Um outro nome usado para esta coleção é alfabeto, ou seja, o conjunto de átomos que servirão de base para a construção de cadeias complexas, inclusive os textos usuais. Entre os símbolos citados podemos destacar alguns subconjuntos relevantes, tais como:

1. As letras maiúsculas do alfabeto;
2. As letras minúsculas do alfabeto;
3. Os algarismos arábicos;

Estes três gozam de uma propriedade muito importante. Dentre deles os símbolos possuem uma relação de ordem, de tal forma que podemos usar a noção usual de ordenação para letras e algarismos.

Além desses, podemos citar ainda os sinais de pontuação e alguns símbolos com funções especiais, como por exemplo o indicador de final de linha de texto. Os símbolos são sempre apresentados em Python entre aspas simples, para que possam ser diferenciados dos nomes de parâmetros, funções e outros. Por exemplo, a letra ***a*** deve ser referenciada por ***'a' .*** Se assim não fosse, na definição:

```
def f1(a) : return (a,'a')
```

não seria possível identificar que estamos descrevendo um par onde o primeiro termo é o conteúdo do parâmetro **a** e o segundo é o símbolo ***a.***

A coleção total de símbolos forma uma sequência de tal forma que podemos fazer o mapeamento entre a subsequência de números naturais, de 0 a 255 e a sequência de símbolos.

Duas funções básicas permitem que se faça a conversão entre as duas sequências:

1. A função ***chr*** associa um número natural no intervalo [0,255] com o caractere correspondente. Por exemplo, `chr(64)` = '@' e `chr(38)` = '&'.
2. A função ***ord*** faz o mapeamento inverso, ou seja, associa um símbolo com o número natural correspondente. Por exemplo, `ord('?')` = 63 e `ord('%')` = 37.

A tabela a seguir apresenta alguns dos agrupamentos importantes.

		Algarismos		**Letras Maiúsculas**			**Letras Minúsculas**		
32	' '	48	**0**	65	**A**	91	**[**	97	**a**	123	**{**
33	**!**	49	**1**	66	**B**	92	\	98	**b**	124	|
34	"	50	**2**	67	**C**	93	**]**	99	**c**	125	**}**
35	#	51	**3**	68	**D**	94	^	100	**d**	126	~
36	**$**	52	**4**	69	**E**	95	_	101	**e**		
37	**%**	53	**5**	70	**F**	96	`	102	**f**		
38	**&**	54	**6**	71	**G**			103	**g**		
39	\	55	**7**	72	**H**			102	**h**		
40	**(**	56	**8**	73	**I**			105	**i**		
41	**)**	57	**9**	74	**J**			106	**j**		
42	*****	58	**:**	75	**K**			107	**k**		
43	+	59	**;**	76	**L**			108	**l**		
44	**,**	60	<	77	**M**			109	**m**		
45	**-**	61	=	78	**N**			110	**n**		
46	**.**	62	>	79	**O**			111	**o**		
47	/	63	**?**	80	**P**			112	**p**		
		64	**@**	81	**Q**			113	**q**		
				82	**R**			114	**r**		
				83	**S**			115	**s**		
				84	**T**			116	**t**		
				85	**U**			117	**u**		
				86	**V**			118	**v**		
				87	**W**			119	**w**		
				88	**X**			120	**x**		
				89	**Y**			121	**y**		
				90	**Z**			122	**z**		

Além das operações **ord** e **chr**, podemos contar com os operadores relacionais que comparam dois símbolos, como no exemplo:

```
>>> 'c'>'h'
False
>>>
```

Podemos agora construir algumas definições interessantes, conforme se apresenta no quadro a seguir.

```
# verifica se um dado símbolo é letra
def letra(x):
  return maiuscula(x) or minuscula(x)
def maiuscula(x):
  return pertence(x,('A','Z'))
def minuscula(x):
  return pertence(x,('a','z'))
def pertence(x,intervalo):
  (a,b)=intervalo
  return x>=a and x<=b

# verifica se um símbolo é um algarismo
def algarismo(x):
  return pertence(x,('0','9'))

# Associa uma letra minuscula com a sua correspondente maiuscula e mantém o símbolo fornecido nos demais casos
def caps(x):
  if minuscula(x) : return chr(ord(x)-32)
  else : return x

# Determina a posição relativa de uma letra dentro do alfabeto, onde as maiúsculas e minúsculas possuem a mesma posição.
# Outros símbolos devem ser associados ao valor 0 (zero)
def ordAlfa(x):
  if letra(x) : return ord(caps(x))-64
  else : return 0
```

Eis a aplicação em algumas instâncias:

```
>>> letra('a')
True
>>> letra('&')
False
>>> letra('Z')
True
>>> algarismo('a')
False
>>> algarismo('7')
```

```
True
>>> caps('a')
'A'
>>> caps('Z')
'Z'
>>> caps('#')
'#'
>>> ordAlfa('a')
1
>>> ordAlfa('Z')
26
>>> ordAlfa('$')
0
```

Eis mais alguns exemplos de funções com o tipo char.

```
# verifica se um símbolo é uma letra vogal
def vogal(x):
  vogais = ['A','E','I','O','U']
  return letra(x) and ocorre(caps(x),vogais)
def ocorre(x,xs):
  def ou(x,y): return x or y
  return reduce(ou,[x==y for y in xs])

# verifica se um símbolo é uma letra consoante
def consoante(x):
  return letra(x) and not(vogal(x))

# Descreve uma lista de pares onde o primeiro termo é um
número entre 0 e 255 e o segundo o caracter correspondente.
# O intervalo desejado é informado por um par de valores no
intervalo 0 a 255.
def tabOrdChr(intervalo):
  (i,f) = intervalo
  return [(x,chr(x)) for x in range(i,f+1)]


>>> vogal('a')
True
>>> vogal('U')
True
>>> vogal('x')
False
>>> consoante('f')
True
>>> consoante('e')
False
>>> consoante('2')
```

```
False
>>> tabOrdChr((65,70))
[(65, 'A'), (66, 'B'), (67, 'C'), (68, 'D'), (69, 'E'), (70,
'F')]
```

15.3. O TIPO STRING: Podemos agrupar átomos do tipo caractere para formar o que denominamos de cadeia de caracteres. Um mecanismo de agregação possível é a lista. Podemos por exemplo, escrever em Python a lista ['c','i','d','a','d','a','n','i','a'] para representar a palavra ***cidadania.***

Assim procedendo podemos escrever qualquer texto, uma vez que um texto não é nada mais nada menos que uma longa cadeia de símbolos envolvendo letras e sinais de pontuação. A separação de palavras é obtida pelo uso de um caractere especial, denominado de ***espaço***, que tem representação interna igual a 32. Por exemplo, para representar a expressão "Vamos nessa!", usamos a lista:

['V','a','m','o','s',' ','n','e','s','s','a','!'].

Uma boa notícia é que o Python nos proporciona uma maneira mais amistosa para tratar com cadeias de caracteres. Em Python podemos representa-las envolvendo a cadeia por aspas (duplas ou simples), o que por certo, além de mais elegante e legível, no poupa trabalho. Por exemplo, a cadeia acima representada por ser também escrita na seguinte forma:

"Vamos nessa!"

A uma lista de caracteres o Python associa um tipo sequencial denominado ***string***. Assim, as duas representações acima são semelhantes porém não idênticas, e há funções que possibilitam a conversão entre eles, como mostrado abaixo:

```
>>> list('Vamos nessa!')
['V', 'a', 'm', 'o', 's', ' ', 'n', 'e', 's', 's', 'a', '!']
>>> ''.join(['V', 'a', 'm', 'o', 's', ' ', 'n', 'e', 's',
's', 'a', '!'])
'Vamos nessa!'
>>>
```

Um outro bom lembrete é que com cadeias de caracteres pode-se usar sobre elas todas as operações que sabemos até agora sobre listas.

Por exemplo, podemos construir uma função para contar a quantidade de vogais existente em uma cadeia.

```
# conta as vogais numa cadeia de caracteres
def contaVogais(xs):
  return len([x for x in xs if vogal(x)])
```

```
>>> contaVogais('Vamos nessa!')
4
>>>
```

15.4. FUNÇÕES BÁSICAS PARA O TIPO STRING: Uma cadeia de caracteres herda todas as operações que já apresentamos para as listas. Além dessas, podemos contar ainda com algumas operações que apresentamos a seguir:

`xs.split()` - Associa uma cadeia de caracteres `xs` com a lista de *palavras* nela contida. Entende-se por *palavra* um agrupamento de símbolos diferentes do símbolo *espaço* (*" "*).

```
>>> "Se temos de aprender a fazer, vamos aprender fazendo!".split()
['Se', 'temos', 'de', 'aprender', 'a', 'fazer,', 'vamos', 'aprender', 'fazendo!']
>>>
```

Podemos observar que os símbolos usuais de separação (por exemplo, "**,**" e "**!**" são considerados como parte das *palavras*. Vamos agora construir uma função que considere os símbolos usuais de separação, além do espaço.

```
# A função palavras
def palavras(xs):
  return ['' .join(list(takewhile(letra,x))) for x in xs.split()]

>>> palavras("Se temos de aprender a fazer, vamos aprender fazendo!")
['Se', 'temos', 'de', 'aprender', 'a', 'fazer', 'vamos', 'aprender', 'fazendo']
>>> palavras("jose123 maria456 joana!!!")
['jose', 'maria', 'joana']
>>> "jose123 maria456 joana!!!".split()
['jose123', 'maria456', 'joana!!!']
>>>
```

De fato, a função **palavras** trata a questão a que nos propúnhamos, abandonar os símbolos de pontuação. Acontece que ela abandona muito mais que isso, como podemos ver no exemplo, onde a cadeia "jose123" perde o seu sufixo numérico, tornando-se apenas "jose".

Vamos construir uma nova função então onde isso possa ser resgatado.

```
# A função palavras1
def palavras1(xs):
  def alfa(x): return letra(x) or algarismo(x)
  return   [   ''.join(list(takewhile(alfa,x)))   for   x   in
xs.split() ]

>>> palavras1("jose123 maria456 joana!!!")
['jose123', 'maria456', 'joana']
>>> palavras1("x123 y456 aux@#")
['x123', 'y456', 'aux']
>>>
```

Bom, parece que agora temos uma solução adequada.

16. O PARADIGMA RECURSIVO

16.1. INTRODUÇÃO: Como já falamos anteriormente, existem várias maneiras de definir um conceito. A essas maneiras convencionamos chamar de paradigmas. Aqui trataremos de mais um destes, o paradigma recursivo. Dizer que trataremos de "mais um" é simplificar as coisas, na verdade este paradigma é um dos mais ricos e importantes para a descrição de computações. O domínio deste paradigma é de fundamental importância para todo aquele que deseja ser um *expert* em Programação de Computadores enquanto ciência e tecnologia.

De uma maneira simplificada podemos dizer que o núcleo deste paradigma consiste em descrever um conceito de forma recursiva. Isto equivale a dizer que definiremos um conceito através do uso do próprio conceito. Apesar de disto parecer muito intrigante, não se assuste, aos poucos, quando esboçarmos melhor a ideia ela se mostrará precisa, simples e poderosa.

Vamos pensar num conceito bem corriqueiro, uma escada. Vamos definir o que entendemos por uma escada? Podemos inicialmente dizer que uma escada é uma sequência de degraus. Mas será que podemos descrever uma escada de uma outra maneira, usando a noção de recursão? A resposta é sim. Podemos definir uma escada assim:

Uma escada é igual a um degrau seguido de uma escada.

Fácil não é? Será que isto basta? Onde está o truque? Parece que estamos andando em círculo, não é mesmo? Para entender melhor vamos discutir a seguir alguns elementos necessários para a utilização correta da recursão na definição de novas funções.

16.2. DESCRIÇÃO RECURSIVA DE UM CONCEITO FAMILIAR: Antes de avançar em nossa discussão vamos apresentar mais um exemplo. Desta vez usaremos um que é bastante familiar para alunos de ciências exatas. Estamos falando da descrição do fatorial de um número. Já vimos neste curso uma forma de descrever este conceito dentro do Python quando estudamos o paradigma aplicativo. Na oportunidade usamos a seguinte descrição:

*O fatorial de um número **n** é o produto de todos os números compreendidos no intervalo que vai de um até **n**.*

Ou ainda em notação mais formal:

$$n ! = 1 \times 2 \times 3 \times ... \times n$$

Em Python, como já vimos, temos a seguinte definição:

```
# fatorial x = produto da lista [1,2,3,...,n]
def fat(n): return product(range(1,n+1))

>>> fat(5)
120
>>> fat(0)
1
>>> fat(1)
1
```

Há uma outra forma de definir, também familiar aos alunos do primeiro ano universitário:

*O fatorial de um número natural **n** é igual ao produto deste número pelo fatorial de seu antecessor.*

Novamente, sendo mais formal, podemos escrever:

$$n ! = n \times (n - 1) !$$

E em Python, como ficaria?
Uma possível definição seria:

```
# definição recursiva de fatorial
def fat(n): return n*fat(n-1)

>>> fat(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File
"/Users/alberto/Dropbox/IComp/disciplinas/ic/LivrosTutoriaisE
tc/PythonCode/s16.py", line 7, in fat
    return n*fat(n-1)
...
RecursionError: maximum recursion depth exceeded
>>>
```

Bom, parece que houve um pequeno problema com nossa definição. A avaliação de `fat(5)` produziu uma situação de erro. Vamos deixar para entender melhor este erro depois. Por enquanto já podemos adiantar que ele foi provocado por um pequeno esquecimento de nossa parte. Na verdade a nossa definição

recursiva para fatorial estava incompleta. A que exibimos só se aplica aos naturais maiores que zero. A definição do fatorial de zero não é recursiva, ela é independente:

O fatorial de zero é igual a 1

Temos então duas definições para fatorial e precisamos integrá-las. Vejamos uma tentativa:

O Fatorial de um número natural n é:

1. *igual a 1 se n=0;*
2. *igual ao produto deste número pelo fatorial de seu antecessor, se n > 0*

Vamos ver como essa integração pode ser feita em Python. Podemos de imediato observar que trata-se de uma definição condicional e logo nos vem a lembrança de que nossa linguagem possui um mecanismo, as expressões condicionais.

```
# definição recursiva de fatorial (corrigida)
def fat(n):
  if n==0 : return 1
  else : return n*fat(n-1)

>>> fat(5)
120
>>> fat(20)
2432902008176640000
>>> fat(0)
1
```

Pelo visto agora deu tudo certo.

16.3. ELEMENTOS DE UMA DESCRIÇÃO RECURSIVA: Em uma descrição recursiva devemos ter em conta certo elementos importantes. É fundamental que todos eles sejam contemplados para que nossas descrições estejam corretas. O exemplo anteriormente apresentado é suficiente para ilustrar todos eles. Vamos então discuti-los:

Definição geral : Toda definição recursiva tem duas partes, uma delas se aplica a um valor qualquer do domínio do problema, que denominamos de “geral”. Ela tem uma característica muito importante, o conceito que está sendo definido deve ser utilizado. Por exemplo, para definir o fatorial de *n*, usamos o fatorial do *antecessor* de *n*. Observe aqui, entretanto que o mesmo conceito foi utilizado, mas

não para o mesmo valor. Aplicamos o conceito a um valor mais simples, neste caso o antecessor de n.

Definição independente : A outra parte da definição é destinada ao tratamento de um valor tão simples que a sua definição possa ser dada de forma independente. Este elemento é também conhecido como “base” da recursão. No caso do fatorial, o valor considerado é o zero.

Obtenção de valores mais simples : Para aplicar o conceito a um valor mais simples precisamos de uma função que faça este papel. No caso do fatorial, usamos a subtração de n por 1, obtendo assim o antecessor de n. Em cada caso, dependendo do domínio do problema e do problema em si, precisaremos encontrar a função apropriada.

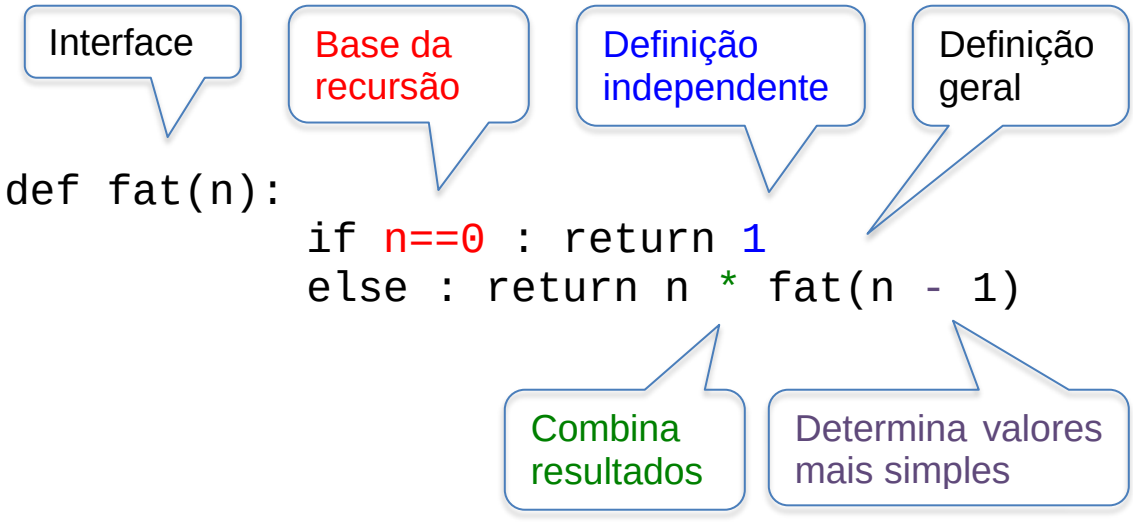
Função auxiliar : Na definição geral, para obter um valor usando o valor considerado e o valor definido recursivamente, em geral faz-se necessário o uso de uma função auxiliar. Algumas vezes essa função pode ser originada a partir de um conceito aplicável a dois elementos e que desejamos estender aos elementos de uma lista. Um exemplo é o caso da somatória dos elementos de uma lista, como veremos adiante. No caso do fatorial esta função é a multiplicação.

Garantia de atingir o valor independente : É fundamental que a aplicação sucessiva da função que obtém valores mais simples garanta a determinação do valor mais simples. Por exemplo, no caso do fatorial, sabemos que aplicando a subtração sucessivas vezes produz a sequência:

n, (n-1), (n-2), ..., 0

Esta condição é fundamental para garantir que ao avaliarmos uma expressão atingiremos a base da recursão.

Voltemos à definição do fatorial para destacarmos os elementos acima citados, como podemos observar no quadro esquemático a seguir:



16.4. AVALIANDO EXPRESSÕES: A esta altura dos acontecimentos a curiosidade sobre como avaliar expressões usando conceitos definidos recursivamente já deve estar bastante aguçada. Não vamos, portanto retardar mais essa discussão. Apresentamos a seguir um modelo bastante simples para que possamos entender como avaliar expressões que usam conceitos definidos recursivamente. Novamente não precisaremos entender do funcionamento interno de um computador nem da maneira como uma determinada implementação de Python foi realizada. Basta-nos o conceito de redução que já apresentamos anteriormente.

Relembremos o conceito de redução. O avaliador deve realizar uma sequência de passos substituindo uma expressão por sua definição, até que se atinja as definições primitivas e os valores possam ser computados diretamente.

Vamos aplicar então este processo para realizar a avaliação da expressão

```
fat(5)
```

passo	Redução	Justificativa
0	`fat(5)`	expressão proposta
1	`5*fat(4)`	substituindo fat por sua definição geral
2	`5*(4*fat(3))`	Idem
3	`5*(4*(3*fat(2)))`	Idem
4	`5*(4*(3*(2*fat(1))))`	Idem
5	`5*(4*(3*(2*(1*fat(0)))))`	Idem
6	`5*(4*(3*(2*(1*1))))`	usando a definição específica
7	`5*(4*(3*(2*1)))`	usando a primitiva de multiplicação
8	`5*(4*(3*2))`	Idem
9	`5*(4*6)`	Idem
10	`5*24`	Idem
11	`120`	Idem

Simples, não? É assim mesmo, bem simples. A cada passo vamos substituindo uma expressão por outra até que nada mais possa ser substituído. O resultado surgirá naturalmente. Mais tarde voltaremos ao assunto.

16.5. RECURSÃO EM LISTAS: A esta altura já estamos certos que o uso de lista é indispensável para escrever programas interessantes. Em vista disso, nada mais óbvio que perguntar sobre o uso de recursão em listas. Veremos que o uso de definições recursivas em listas produz descrições simples, precisas e elegantes.

Já está na hora de alertar que os valores sobre os quais aplicamos os conceitos que queremos definir recursivamente possuem uma característica importantíssima, eles em si são recursivos.

Por exemplo, qualquer valor pertencente aos naturais pode ser descrito a partir da existência do zero e da função sucessor `suc()`. Vejamos como podemos obter o valor 5:

5 = suc(suc(suc(suc(suc(0)))))

As listas são valores recursivos. Podemos descrever uma lista da seguinte maneira:

Uma lista é:

1. a lista vazia;
2. um elemento seguido de uma lista.

Esta natureza recursiva das lista nos oferece uma oportunidade para, com certa facilidade, escrevermos definições recursivas. A técnica consiste basicamente em:

1. Obter a definição geral: isto consiste em identificar uma operação binária simples que possa ser aplicada a dois valores. O primeiro deles é o primeiro elemento (`head`) da lista e o outro é um valor obtido pela aplicação do conceito em definição ao resto da lista (`tail`);
2. Obter a definição independente;
3. Garantir que a aplicação sucessiva do `tail` levará à base da recursão.

Exemplo 01 - Descrever a somatória (somat) dos elementos de uma lista.

Solução : Podemos pensar da seguinte maneira: a somatória dos elementos de uma lista é igual à soma do primeiro elemento da lista como a somatória do resto da lista. Além disso, a somatória dos elementos de uma lista vazia é igual a zero.

```
# definição recursiva da somatória dos elementos de uma lista
def somat(xs):
  if len(xs)==0 : return 0
  else : return head(xs)+somat(tail(xs))

>>> somat([4,5,2,7,9])
27
>>> somat(range(1,11))
55
```

```
>>> somat(range(100,0,-1))
5050
```

Exemplo 02 - Descrever a função que determina o elemento de valor máximo uma lista de números.

Solução: O máximo de uma lista é o maior entre o primeiro elemento da lista e o máximo aplicado ao resto da lista. Uma lista que tem apenas um elemento tem como valor máximo o próprio elemento.

```
# definição recursiva do máximo de uma lista
def maximo(xs):
  if len(tail(xs))==0 : return head(xs)
  else : return maior(head(xs),maximo(tail(xs)))
def maior(x,y):
  if x>y : return x
  else : return y

>>> maximo([4,6,7,89,32,45,98,65,31])
98
>>> maximo([1,4,-1,123,321]+[300,400])
400
>>> maximo([10])
10
>>> maximo([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File
"/Users/alberto/Dropbox/IComp/disciplinas/ic/LivrosTutoriaisE
tc/PythonCode/s16.py", line 23, in maximo
    if len(tail(xs))==0 : return head(xs)
  File
"/Users/alberto/Dropbox/IComp/disciplinas/ic/LivrosTutoriaisE
tc/PythonCode/s12.py", line 61, in head
    def head(xs): return xs[0]
IndexError: list index out of range
>>>
```

Exemplo 03 - Descrever a função que verifica se um dado valor ocorre em uma lista.

Solução : Podemos pensar da seguinte maneira: Um dado elemento k ocorre em uma lista se ele é igual ao primeiro elemento da lista ou se ele ocorre no resto da lista. Em uma lista vazia não ocorrem elementos quaisquer.

```
# descreve a ocorrência de dado k em uma lista xs
def ocorre(k,xs):
  if len(xs)==0 : return False
  else : return ( (k==head(xs)) or ocorre(k,tail(xs)) )
```

```
>>> ocorre(5,[8,65,46,23,99,35])
False
>>> ocorre(5,[8,65,46,5,23,99,35])
True
>>> ocorre(5,[])
False
>>>
```

Exemplo 04 - Descrever a função que obtém de uma lista xs a sublista formada pelos elementos que são menores que um dado k :

Solução : Precisamos descrever uma nova lista, vamos denominá-la de ***menores***, em função de xs e de k. Quem será esta nova lista? Se o primeiro elemento de xs for menor que k, então ele participará da nova lista, que pode ser descrita como sendo formada pelo primeiro elemento de xs seguido dos menores que k no resto de xs. Se por outro lado o primeiro não é menor que k, podemos dizer que a lista resultante é obtida pela aplicação de menores ao resto da lista. Novamente a base da recursão é definida pela lista vazia, visto que em uma lista vazia não ocorrem elementos menores que qualquer k.

```
# define a lista de menores que um dado elemento em uma lista
dada
def menores(k,xs):
  if len(xs)==0 : return xs
  elif head(xs) < k : return [head(xs)] + menores(k,tail(xs))
  else : return menores(k,tail(xs))

>>> menores(23,[8,65,46,5,23,99,35])
[8, 5]
>>> menores(46,[8,65,46,5,23,99,35])
[8, 5, 23, 35]
>>> menores(5,[])
[]
>>>
```

16.6. EXPLORANDO REUSO: Segundo o Professor George Polya, após concluir a solução de um problema, devemos levantar questionamentos a respeito das possibilidades de generalização da solução obtida. Vamos explorar um pouco esta abordagem após a solução do problema a seguir.

Sub-lista de números pares: Dada uma lista xs, desejamos descrever uma sublista de xs formada apenas pelos números pares existentes em xs.

Solução: Devemos considerar, como no problema de encontrar a sublista dos menores, a existência de suas situações:

1. O primeiro elemento da lista é um número par, neste caso a sublista resultante é dada pela junção do primeiro elemento com a sublista de pares existente no resto da lista.
2. O primeiro não é par. Neste caso a sublista de pares em xs é obtida pela seleção dos elementos pares do resto de xs.

Concluindo, tomemos como base da recursão a lista vazia, que obviamente não contém qualquer número.

Eis a solução em Python:

```
# sublista de números pares
def slpares(xs):
  if len(xs)==0 : return xs
  elif even(head(xs)) : return [head(xs)] + slpares(tail(xs))
  else : return slpares(tail(xs))
def even(x): return x%2==0

>>> slpares([1,2,3,4,5,6,7,8,9,10])
[2, 4, 6, 8, 10]
>>> slpares([3,25,4,16,7,10])
[4, 16, 10]
>>> slpares([])
[]
>>>
```

Vamos agora, seguindo as orientações do mestre Polya, buscar oportunidades de generalização para esta função. Podemos fazer algumas perguntas do tipo: como faria uma função para determinar a sublista dos números ímpares a partir de uma dada lista? E se quiséssemos a sublista dos primos? E que tal a dos múltiplos de 5?

Uma breve inspeção na solução acima nos levaria a entender que a única diferença entre as novas funções e a que já temos é a função que verifica se o primeiro elemento satisfaz uma propriedade, no caso presente a de ser um número par (even), conforme destacamos a seguir:

```
# sublista de números pares
def slpares(xs):
  if len(xs)==0 : return xs
  elif even(head(xs)) : return [head(xs)] + slpares(tail(xs))
  else : return slpares(tail(xs))
def even(x): return x%2==0
```

```
# sublista de números impares
def slimpares(xs):
  if len(xs)==0 : return xs
  elif odd(head(xs)) : return [head(xs)] + slimpares(tail(xs))
  else : return slimpares(tail(xs))
def odd(x): return not even(x)

# sublista de números primos
def slprimos(xs):
  if len(xs)==0 : return xs
  elif primo(head(xs)) : return [head(xs)] + slprimos(tail(xs))
  else : return slprimos(tail(xs))
def primo(x):
  if even(x) : return False
  elif x==1 : return True
  else : return [i for i in range(3,x,2) if x%i==0] == []
```

Isto nos sugere que a função avaliadora pode ser um parâmetro. Pois bem, troquemos então o nome da função por um nome mais geral e adicionemos à sua interface mais uma parâmetro. Este parâmetro, como sabemos, deverá ser do tipo boolean. Vejamos então o resultado:

```
# sublista de elementos de xs que satisfazem a propriedade prop
def sublista(prop,xs):
  if len(xs)==0 : return xs
  elif prop(head(xs)) : return [head(xs)] + sublista(prop,tail(xs))
  else : return sublista(prop,tail(xs))

>>> slpares([1,2,3,4,5,6,7,8,9,10])
[2, 4, 6, 8, 10]
>>> slimpares([1,2,3,4,5,6,7,8,9,10])
[1, 3, 5, 7, 9]
>>> slprimos([1,2,3,4,5,6,7,8,9,10])
[1, 3, 5, 7]
>>> sublista(even,[1,2,3,4,5,6,7,8,9,10])
[2, 4, 6, 8, 10]
>>> sublista(odd,[1,2,3,4,5,6,7,8,9,10])
[1, 3, 5, 7, 9]
>>> sublista(primo,[1,2,3,4,5,6,7,8,9,10])
[1, 3, 5, 7]
>>> sublista(notamin,[1,2,3,4,5,6,7,8,9,10])
[5, 6, 7, 8, 9, 10]
>>>
```

16.7. ORDENAÇÃO: Voltemos então ao problema de ordenar os elementos de uma lista, para o qual já discutimos uma solução no paradigma aplicativo.

Dada uma lista xs desejamos descrever sua ordenação.

Vamos começar propondo e resolvendo um problema mais simples.

Inserção ordenada: Dada uma lista ordenada xs e um elemento k desejamos descrever uma lista a partir de xs, na qual esteja incluído o valor k, com a condição de que a nova lista também esteja ordenada.

Solução: Voltemos a nossa estratégia para obter soluções recursivas. Aqui também temos dois casos:

1. O valor k é menor que o primeiro da lista xs, neste caso a lista resultante é descrita pela junção de k com a lista xs;
2. O valor k é maior ou igual ao primeiro elemento da lista xs, neste caso a lista resultante é descrita pela junção do primeiro da lista xs com a lista obtida pela inserção ordenada de k no resto da lista xs.

```
# inserção ordenada de um valor k em uma lista ordenada (não
decrescente)
def insord(k,xs):
  if len(xs)==0 : return [k]
  elif k < head(xs) : return [k]+xs
  else : return [head(xs)] + insord(k,tail(xs))

>>> insord(5,[0,2,4,6,8,10])
[0, 2, 4, 5, 6, 8, 10]
>>> insord(5,[10,15,20,25,30])
[5, 10, 15, 20, 25, 30]
>>> insord(5,[-10,-5,0,10,15,20,25,30])
[-10, -5, 0, 5, 10, 15, 20, 25, 30]
>>> insord(5,[-10,-5,0,5,10,15,20,25,30])
[-10, -5, 0, 5, 5, 10, 15, 20, 25, 30]
>>> insord(5,[])
[5]
>>>
```

Agora já podemos voltar à ordenação. Vamos à solução.

Solução: A ordenação não decrescente de uma lista xs qualquer é igual à inserção ordenada do primeiro da lista na ordenação do resto da lista.

```
# ordenação de uma lista
def ordena(xs):
  if len(xs)==0 : return xs
  else : return insord(head(xs),ordena(tail(xs)))

>>> ordena([3, 4, 50,30,20,34,15])
[3, 4, 15, 20, 30, 34, 50]
>>> ordena([3,4,50,-30,10,30,20,34,0,15])
[-30, 0, 3, 4, 10, 15, 20, 30, 34, 50]
>>>
```

16.8. DIVISÃO E CONQUISTA: Muitos problemas possuem soluções mais facilmente descritas, algumas até mais eficientes, quando quebramos o problema em partes menores, descrevemos a solução de cada parte e depois combinamos as soluções parciais para obter a solução completa. Este método é denominado de "divisão e conquista". Basicamente buscamos encontrar instâncias do problema onde a solução seja imediata. Nesta seção veremos alguns exemplos desta abordagem. O primeiro deles, a pesquisa binária, trata da busca de um elemento em uma lista ordenada. Os outros dois, *mergesort* e *quicksort*, apresentam soluções alternativas para a ordenação de uma lista.

16.8.1. PESQUISA BINÁRIA - Voltemos à verificação da ocorrência de um elemento a uma lista, segundo a definição que apresentamos para a função ocorre. Podemos constatar que para avaliar expressões onde o elemento procurado não ocorre na lista, o avaliador de expressões precisará fazer uma quantidade de comparações igual ao comprimento da lista considerada. Na média de um conjunto de avaliações, considerando as avaliações de expressões em que o elemento procurado está na lista, e que a cada vez estaremos procurando por um elemento distinto, teremos um número médio de comparações da ordem de (**n/2**). Se n for muito grande ficaremos assustados com o número de comparações. Por exemplo, para uma lista de 1000000 (um milhão) de elementos, em média teremos que fazer 500 mil comparações.

Se pudermos garantir que a lista está ordenada, então podemos fazer uso de uma estratégia já discutida anteriormente para reduzir este número. Falamos da árvore binária de pesquisa. A estratégia que usaremos consiste de, a cada passo de redução, abandonarmos metade da lista considerada a partir da comparação de k com o elemento que se encontra na metade da lista. Se o elemento buscado (k) for igual ao elemento central, então o processo de avaliação está encerrado. Quando isto não ocorre, devemos então escolher em qual lista devemos procurá-lo. Quando ele é menor que o elemento central devemos buscá-lo na sublista que antecede o central, caso contrário devemos buscá-lo na sublista dos seus sucessores. Novamente a base da recursão é determinada pela lista vazia.

Nesta abordagem, a cada escolha abandonamos metade da lista restante. Desta forma, o número de comparações é dado pelo tamanho da sequência:

n/1, n/2, n/4, ... , n/n

Para simplificar a análise podemos escolher um n que seja potência de 2. Neste caso podemos assegurar que o comprimento da seqüência é dado por:

Log n na base 2

Voltando então ao número de comparações necessárias para localizar um elemento, podemos constatar que em uma lista com 1 milhão de elementos, ao invés das 500 mil comparações da solução anterior, precisaremos no pior caso, de apenas 20.

Vamos então à codificação em Python:

```
# pesquisa binária
def pesqbin(k,xs) :
  if len(xs)==0 : return False
  p       = len(xs)//2
  menores = take(p,xs)
  maiores = tail(drop(p,xs))
  pivot   = head(drop(p,xs))
  if k == pivot : return True
  elif k<pivot : return pesqbin(k,menores)
  else : return pesqbin(k,maiores)

>>> ocorre(50,[1,19,23,35,60,72])
False
>>> ocorre(50,[1,19,23,35,50,60,72])
True
>>>
```

16.8.2. MERGESORT - Existem outras maneiras de se descrever a ordenação de uma lista. Uma delas, denominada ***mergesort***, se baseia na intercalação de duas listas já ordenadas. Vejamos uma descrição abstrata.

Intercalação: Antes de ver o mergesort podemos apresentar uma versão recursiva para a intercalação de duas listas em ordem não decrescente.

Solução: A intercalação de duas listas ordenadas ***xs*** e ***ys*** pode ser descrita através de dois casos:

1. se o primeiro elemento de ***xs*** é menor que o primeiro elemento de ***ys*** então a intercalação é dada pela junção do primeiro elemento de ***xs*** com a intercalação do resto de ***xs*** com ***ys***;

2. caso contrário, a intercalação é descrita pela junção do primeiro elemento de ***ys*** com a intercalação do resto de ***ys*** com ***xs***

```
# Intercala duas listas em ordem não decrescente
def intercala(xs,ys):
  if len(xs)==0 or len(ys)==0 : return xs+ys
  elif head(xs)<=head(ys) : return [head(xs)] + intercala(tail(xs),ys)
  else : return [head(ys)] + intercala(xs,tail(ys))

>>> intercala([1,3,5,7,9,10],[0,2,4,6,8,10])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10]
>>> intercala([1,3,5,7,9,10],[])
[1, 3, 5, 7, 9, 10]
>>> intercala([],[])
[]
>>>
```

Solução : A ordenação de uma lista por mergesort é igual à intercalação do mergesort da primeira metade da lista com o mergesort da segunda metade. Esta solução explora a noção de árvore binária. Neste caso, a lista original é dividida em 2 partes, cada uma delas em outras duas e assim sucessivamente até que esta quebra não seja mais possível. A figura Fig. 9.1 ilustra o processamento da ordenação de uma lista.

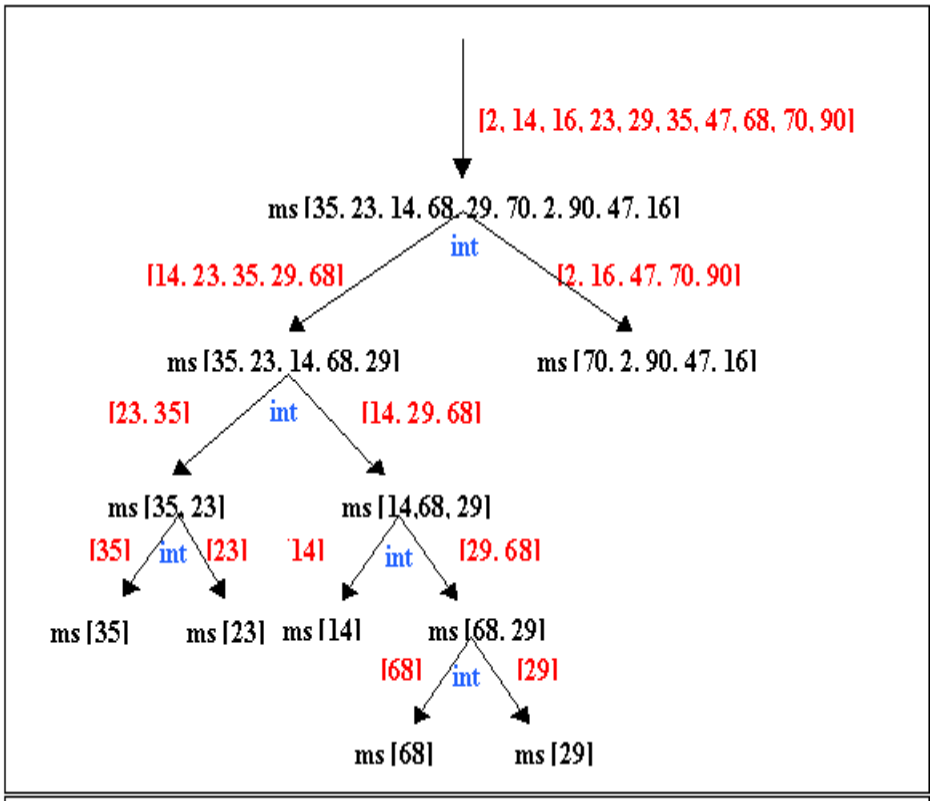


Fig 9.1 : Desenvolvimento da ordenação de uma lista utilizando o mergesort. As lista em preto são as entradas para a função e as listas em vermelho são os valores retornados. Em azul indicamos a função de intercalação (int) aplicada ao resultados de mergesort.

Vejamos então como fica a codificação em Python.

```
# ordena uma lista pela intercalação da ordenação de suas
duas metades
def mergesort(xs):
  if len(tail(xs))==0 : return xs
  k = len(xs)//2
  m = take(k,xs)
  n = drop(k,xs)
  return intercala(mergesort(m),mergesort(n))

>>> mergesort([1,3,-5,7,9,100,20,4,65,8,10])
[-5, 1, 3, 4, 7, 8, 9, 10, 20, 65, 100]
>>> mergesort([1,3,-5,7,0,100,20,4,65,18])
[-5, 0, 1, 3, 4, 7, 18, 20, 65, 100]
>>> mergesort([1])
[1]
>>> mergesort([])
[]
```

16.8.3. QUICKSORT - Existe uma maneira muito famosa de resolver o mesmo problema, usando ainda a noção de divisão e conquista, muito parecida com o mergesort. Implementações desta solução reduzem sensivelmente o número de comparações necessárias e são portanto muito utilizadas.

Solução: Na versão usando o mergesort dividíamos a instância original exatamente ao meio. Nesta proposta vamos dividir também em duas, mas com seguinte critério: a primeira com os elementos menores que um elemento qualquer da lista e a segunda com os elementos maiores ou iguais a ele. Este elemento é denominado ***pivot*** e existem várias formas de escolhê-lo. A melhor escolha é aquela que produzir as sublistas com comprimentos bem próximos, o que repercutirá no desempenho da avaliação. Aqui nos limitaremos a escolher como pivot o primeiro elemento da lista. Assim sendo, após obter a ordenação das duas listas, basta juntar a ordenação da primeira, com o pivot e finalmente com a ordenação da segunda. A figura Fig. 9.2 ilustra a aplicação do quicksort a uma instância do problema.

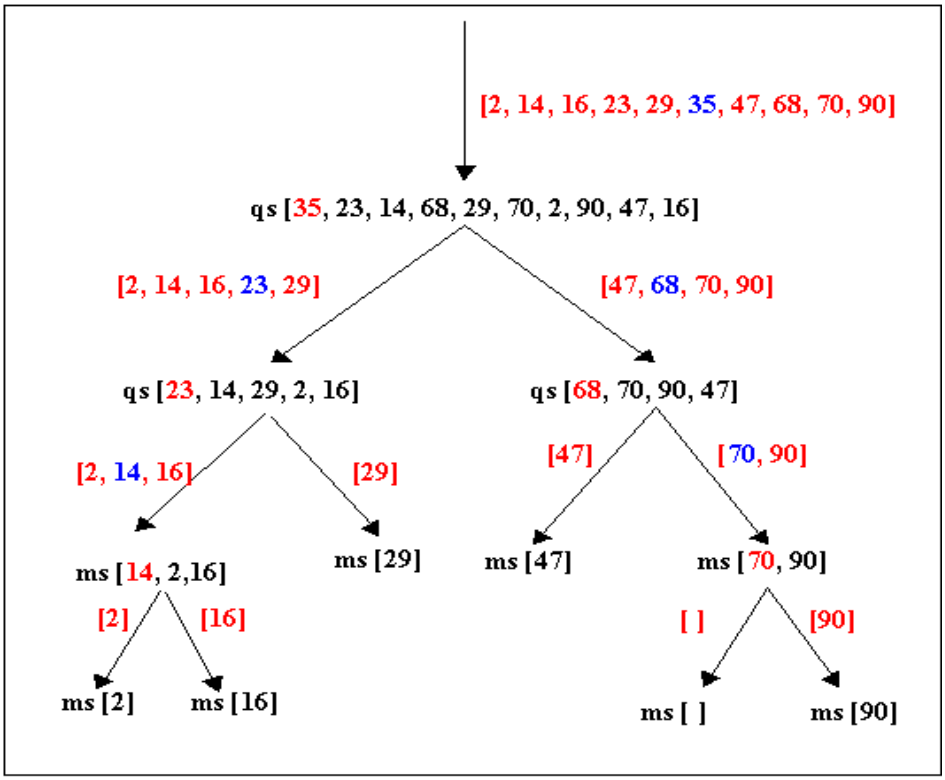


Fig 9.2 : Desenvolvimento da ordenação de uma lista utilizando o quicksort. As listas em preto são as entradas para a função e as listas em vermelho são os valores retornados. Nas listas de entrada indicamos o pivot em vermelho, nas listas de retorno o pivot está indicado em azul.

Vejamos a codificação em Python:

```
# quicksort cria e ordena sublistas dos menores e dos maiores
que pivot
def quicksort(xs):
  if  len(xs)==0 or len(tail(xs))==0 : return xs
  pivot = head(xs)
  def men_pivot(x): return x<pivot
  def mig_pivot(x): return x>=pivot
  return  ( quicksort(sublista(men_pivot,tail(xs)))
          + [pivot]
          + quicksort(sublista(mig_pivot,tail(xs))) )

>>> quicksort([1,3,-5,7,9,100,20,4,65,8,10])
[-5, 1, 3, 4, 7, 8, 9, 10, 20, 65, 100]
>>> quicksort([1,3,-5,7,0,100,20,4,65,18])
[-5, 0, 1, 3, 4, 7, 18, 20, 65, 100]
>>> quicksort([15])
[15]
>>> quicksort([])
[]
>>>
```

16.9. CADEIAS DE CARACTERES: As cadeias de caracteres, como já vimos, são objetos sequenciados como as listas, portanto o uso de recursão com cadeias segue as mesma recomendações. Para ilustrar vamos apresentar alguns exemplos.

Exemplo 01 - [Palíndromo] Dada uma cadeia de caracteres verifique se é um palíndromo. Segundo o dicionário, um palíndromo é uma frase ou palavra, que não importando o sentido que se lê, significa a mesma coisa. Por exemplo, “SOCORRAM ME SUBI NO ONIBUS EM MARROCOS”. Vejam que a quantidade de espaços, os separadores e os términos de palavra não são considerados. Aqui vamos tratar a questão de forma simplificada, os separadores serão tratados como caracteres comuns.

Solução: Neste caso, é importante observar que podemos olhar a cadeia como sendo formada por pares de valores equidistantes dos extremos. Um cadeia é palíndromo se os seus extremos são iguais e o meio da lista é um palíndromo. A base da recursão são as cadeias vazias ou aquelas com apenas um elemento.

Vejamos então a codificação em Python e a avaliação para algumas instâncias.

```
# Verifica se uma sentença é Palíndromo
def palindromo(xs):
  if len(xs)==0 or len(tail(xs))==0 : return True
  else : return head(xs)==last(xs) and palindromo(meio(xs))
def meio(xs): return init(tail(xs))

>>> palindromo('amoroma')
True
>>> palindromo('amoaroma')
False
>>> palindromo('socorrammesubinoonibusemmarrocos')
True
>>>
```

Exemplo 02 - [Prefixo] Dadas duas cadeias de caracteres verifique se a primeira é idêntica à subcadeia formada pelos primeiros caracteres da segunda. Por exemplo, "aba" é prefixo da cadeia "abacaxi" e "pre" é prefixo de "prefixo".

Solução: De imediato podemos dizer que uma cadeia xs é prefixo de uma cadeia ys quando seus primeiros elementos são iguais e o restante de xs é prefixo do restante de ys. Quanto à base da recursão, temos que considerar duas situações. A primeira tem como base que a cadeia vazia é prefixo de qualquer outra cadeia. A segunda leva em conta que nenhuma cadeia pode ser prefixo de uma cadeia vazia (exceto a cadeia vazia).

Vejamos como fica em Python:

```
# Verifica se uma cadeia xs é prefixo de uma segunda (ys)
def prefixo(xs,ys):
  if len(xs)==0 : return True
  elif len(ys)==0 : return False
  else : return head(xs)==head(ys) and prefixo(tail(xs),tail(ys))

>>> prefixo('aba','abacadraba')
True
>>> prefixo("",'abacadraba')
True
>>> prefixo("pre","prefixo")
True
>>> prefixo("prefixo",'pre')
False
>>> prefixo("prefixo",'')
False
>>>
```

Exemplo 03 - [Casamento de Padrão] Verificar se uma cadeia satisfaz um determinado padrão é um processamento muito útil e constantemente realizado na prática da computação. Aqui nos ateremos a uma forma simplificada deste problema que consiste em verificar se uma cadeia é ***subcadeia*** de outra.

Solução: Uma rápida inspeção nos leva à constatação de que o problema anterior é parecido com este, exceto pelo fato de que a primeira cadeia pode ocorrer em qualquer lugar da segunda. Podemos dizer então que a primeira cadeia ocorre na segunda se ela é um prefixo da primeira ou se ela ocorre no resto da segunda.

Vejamos como fica em Python:

```
# Verifica se uma cadeia xs é subcadeia de uma outra (ys)
def subcadeia(xs,ys):
  if len(ys)==0 or len(tail(ys))==0 : return False
  else : return prefixo(xs,ys) or subcadeia(xs,tail(ys))

>>> subcadeia('','prefacio')
True
>>> subcadeia('pre','prefacio')
True
>>> subcadeia('cio','prefacio')
True
>>> subcadeia('efa','prefacio')
True
>>> subcadeia('acido','prefacio')
False
>>> subcadeia('efa','')
False
>>>
```