Departamento de Matemática da Universidade de Coimbra		
2010/2011	**Programação Orientadas para os Objectos**	**Projecto 1a**

Cálculo de Expressões Aritméticas

A leitura e posterior cálculo de uma expressão aritmética na notação usual (a notação infixa) é algo um pouco complicado de implementar, isto dado ser necessário considerar algo que não está explícito na própria expressão, as prioridades das operações. Por exemplo: a expressão $x+y\times z$, não é exactamente aquilo que pode parecer à primeira vista, a sua leitura não se pode fazer da forma usual, da esquerda para a direita. Por convenção a multiplicação tem precedência sobre a adição e como tal deve ser feita em primeiro lugar, isto é a expressão deve ser lida como $x+(y\times z)$, com a convenção de que tudo o que está dentro de um par de parêntesis é calculado primeiro.

Fica então evidente que a notação infixa é uma notação ambígua em si mesmo, necessitando de informação externa para poder ser lida de forma correcta. Este facto dificulta o seu tratamento automático.

Há alternativas, as expressões aritméticas podem ser escritas em notação pré-fixa (também designada notação Polaca[2]), ou em notação pós-fixa:

em notação pré-fixa $+\times yzx$

em notação pós-fixa $xyz\times +$

Estas duas formas de escrever as expressões aritméticas são não ambíguas: à esquerda (respectivamente, à direita) de um dado operador estão sempre os seus dois operandos, ou no caso de um operador unário, o seu único operando.

Pretende-se então construir um programa que leia, e calcule o seu valor, expressões aritméticas em notação Polaca (pré-fixa). O procedimento de leitura e cálculo é muito fácil de programar através da utilização de uma árvore binária.

Projecto de Programação Orientada aos Objectos 2010/2011.

1. Implemente, em *C++* a classe Árvore Binária de Palavras:

   Árvore Binária de Palavras = (ÁrvoresBinárias, {obtemRaiz, obtemABesq, obtemABdir, fixaRaiz, fixaABesq, fixaABdir, destroiABesq, destroiABdir, vazia?, travessiaEmOrdem, travessiaPreordem, travessiaPosOrdem})

   Tenha o cuidado de considerar as situações de erro.

2. Construa um programa que receba (leia) uma expressão em notação polaca (notação pré-fixa), a guarde numa árvore binária e calcule e escreva o valor final da expressão através de uma travessia em-ordem.

Documente o seu programa. Tanto em termos de documentação interna, como de documentação externa na forma de um pequeno manual de utilização.

v.s.f.f.

[2]Devido a ter sido proposta pelo matemático Polaco Jan Łukasiewicz

A álgebra "Árvores Binárias" pode ser caracterizada da seguinte forma:

Elementos

$$\text{ÁrvoresBinárias} = \begin{cases} \text{ABVazia}, & \text{árvore binária vazia} \\ (\text{raiz}, \text{ABesq}, \text{ABdir}), & \text{árvores binárias não vazias} \end{cases}$$

Funções internas Por motivos de economia de espaço vai-se escrever AB para ÁrvoresBinárias.

$$\begin{array}{rccl}
\text{obtemRaiz}: & AB & \longrightarrow & Elementos \cup \{erro\} \\
& ab & \longmapsto & \begin{cases} r, & \text{para } ab = (r, abesq, abdir) \\ erro, & \text{para } ab = \text{ABVazia} \end{cases} \\
\text{obtemABesq}: & AB & \longrightarrow & AB \cup \{erro\} \\
& ab & \longmapsto & \begin{cases} abesq, & \text{para } ab = (raiz, abesq, abdir) \\ erro, & \text{para } ab = \text{ABVazia} \end{cases} \\
\text{obtemABdir}: & AB & \longrightarrow & AB \cup \{erro\} \\
& p & \longmapsto & \begin{cases} abdir, & \text{para } ab = (raiz, abesq, abdir) \\ erro, & \text{para } ab = \text{ABVazia} \end{cases} \\
\text{fixaRaiz}: & Elementos \times AB & \longrightarrow & AB \cup \{erro\} \\
& (r, ab) & \longmapsto & \begin{cases} (r, abesq, abdir), & \text{para } ab = (r', abesq, abdir) \\ erro, & \text{para } ab = \text{ABVazia} \end{cases} \\
\text{fixaABesq}: & AB \times AB & \longrightarrow & AB \cup \{erro\} \\
& (abesq, ab) & \longmapsto & \begin{cases} (r, abesq, abdir), & \text{para } ab = (r, abesq', abdir) \\ erro, & \text{para } ab = \text{ABVazia} \end{cases} \\
\text{fixaABdir}: & AB \times AB & \longrightarrow & AB \\
& (abdir, ab) & \longmapsto & \begin{cases} (r, abesq, abdir), & \text{para } ab = (r, abesq, abdir') \\ erro, & \text{para } ab = \text{ABVazia} \end{cases} \\
\text{destroiABesq}: & AB & \longrightarrow & AB \\
& ab & \longmapsto & \begin{cases} (r, ABVazia, abdir), & \text{para } ab = (r, abesq, abdir) \\ erro, & \text{para } ab = \text{ABVazia} \end{cases} \\
\text{destroiABdir}: & AB & \longrightarrow & AB \\
& ab & \longmapsto & \begin{cases} (r, abesq, ABVazia), & \text{para } ab = (r, abesq, abdir) \\ erro, & \text{para } ab = \text{ABVazia} \end{cases} \\
\text{vazia?}: & AB & \longrightarrow & Bool \\
& ab & \longmapsto & \begin{cases} \mathcal{V}, & \text{para } ab = (r, abesq, abdir) \\ \mathcal{F}, & \text{para } ab = \text{ABVazia} \end{cases} \\
\text{emOrdem}: & AB & \longrightarrow & ListaElementos \\
& ab & \longmapsto & \begin{cases} emOrdem(abesq) : r : emOrdem(abdir), & \\ & \text{para } ab = (r, abesq, abdir) \\ listaVazia, & \text{para } ab = \text{ABVazia} \end{cases}
\end{array}$$

No caso da implementação em *C++* devemos ter em conta os construtores e o destrutor, os diferentes construtores terão: zero, um, ou três argumentos.

Não é necessário implementar todas as travessias, basta implementar aquela que é necessária para a resolução do problema em questão.