Programação com Objectos/Projecto de Programação com Objectos/Enunciado do Projecto de 2016-2017

*From Wiki**3*

< Programação com Objectos | Projecto de Programação com Objectos

AVISOS - Avaliação em Época Normal	**[Collapse]**

Esclarecimento de dúvidas:

- Consultar sempre o corpo docente atempadamente: presencialmente ou através do endereço oficial da disciplina [1].
- Não utilizar fontes de informação não oficialmente associadas ao corpo docente (podem colocar em causa a aprovação à disciplina).
- Não são aceites justificações para violações destes conselhos: quaisquer consequências nefastas são da responsabilidade do aluno.

Requisitos para desenvolvimento, material de apoio e actualizações do enunciado (ver informação completa em Projecto de Programação com Objectos):

- O material de apoio é de uso obrigatório e não pode ser alterado.
- Verificar atempadamente (mínimo de 48 horas antes do final de cada prazo) os requisitos exigidos pelo processo de desenvolvimento.

Processo de avaliação (ver informação completa em Avaliação do Projecto):

- Datas: **2016/10/21 12:00** (inicial); **2016/11/21 12:00** (intercalar); **2016/12/09 12:00** (final); **2016/12/09-2016/12/13** (teste prático).
- **A entrega inicial, sendo crucial para o projecto, é obrigatória e sua não realização implica a exclusão da avaliação do projecto e, por consequência, da avaliação da disciplina.**
- Verificar atempadamente (até 48 horas antes do final de cada prazo) os requisitos exigidos pelo processo de avaliação, incluindo a capacidade de acesso ao repositório CVS.
- **Apenas se consideram para avaliação os projectos existentes no repositório CVS oficial.**
- Trabalhos não presentes no repositório no final do prazo têm classificação 0 (zero) (não são aceites outras formas de entrega). Não são admitidas justificações para atrasos em sincronizações do repositório. A indisponibilidade temporária do repositório, desde que inferior a 24 horas, não justifica atrasos na submissão de um trabalho.
- A avaliação do projecto pressupõe o compromisso de honra de que o trabalho correspondente foi realizado pelos alunos correspondentes ao grupo de avaliação.
- **Fraudes na execução do projecto terão como resultado a exclusão dos alunos implicados do processo de avaliação.**

O objectivo do projecto é criar uma aplicação que permite gerir um conjunto de programas e as expressões de que são compostos.

Neste texto, o tipo **negrito** indica um literal (i.e., é exactamente como apresentado); o símbolo ␣ indica um espaço; e o tipo *itálico* indica uma parte variável (i.e., uma descrição).

Estrutura de expressões e programas

Uma expressão é uma representação algébrica de uma quantidade, ou seja, todas as expressões podem ser avaliadas para obteção de um valor.

Material de Uso Obrigatório **[Collapse]**

As bibliotecas **po-uuilib** e o conteúdo inicial do CVS são de **uso obrigatório**:

- po-uuilib (classes de base) media:po-uuilib-201609201009.tar.bz2 (não pode ser alterada)
- pex-core (classes do "core") (via CVS) (deve ser completada)
- pex-app (classes de interacção) (via CVS) (deve ser completada -- os nomes das classes não podem ser alterados)

A máquina virtual, fornecida para desenvolvimento do projecto, já contém todo o material de apoio.

Uso Obrigatório: Repositório CVS

Apenas se consideram para avaliação os projectos existentes no repositório CVS oficial.
Trabalhos não presentes no repositório no final do prazo têm classificação 0 (zero) (não são aceites outras formas de entrega). Não são admitidas justificações para atrasos em sincronizações do repositório. A indisponibilidade temporária do repositório, desde que inferior a 24 horas, não justifica atrasos na submissão de um trabalho.

Contents

Uma expressão pode ser descrita por uma linguagem cuja interpretação resulta directamente, ou no valor (tipicamente, aplicável apenas em situações muito simples), ou na representação dessa expressão como uma estrutura de dados (tal como neste projecto), sobre a qual pode ser então executado o processo de avaliação, para determinação do valor correspondente.

Neste documento, utiliza-se uma linguagem simples e uniforme para descrever qualquer expressão, que pode ser categorizada como expressão primitiva (literais e identificadores) ou como expressão composta (operadores).

Expressões primitivas

As expressões primitivas, ou representam directamente um valor (expressões literais), ou referem um valor (identificadores).

Literais

As expressões literais são de dois tipos: inteiros e cadeias de caracteres.

- Literais inteiros são números decimais não negativos, constituídos por sequências de 1 (um) ou mais dígitos de **0** a **9**. Exemplos: **1**, **23**
- As cadeias de caracteres são delimitadas por aspas (**"**) e podem conter quaisquer caracteres. Exemplos: **"hello, world!\n"**, **"abcd"**.

Identificadores

Um identificador permite referenciar um valor. A avaliação do identificador corresponde à obtenção do seu valor.

São iniciados por uma letra, seguindo-se 0 (zero) ou mais letras ou dígitos. O comprimento do nome é ilimitado e dois nomes são distintos se houver alteração de maiúscula para minúscula, ou vice-versa, de pelo menos um carácter.

Se for referenciado um identificador que não tenha sido definido, o valor devolvido é **0** (zero) e o identificador mantém-se indefinido.

Expressões compostas

Uma expressão composta é constituída por um operador, podendo ter argumentos. Cada argumento é representado por uma expressão.

O valor de uma expressão composta corresponde à avaliação do operador sobre os seus argumentos (no caso dos operadores com argumentos) ou simplesmente do operador (no caso dos operadores sem argumentos). O tipo de argumentos e de resultado depende do operador. Por exemplo, existem operadores que só suportam argumentos do tipo inteiro. A passagem de tipos inapropriados para um operador é um erro (mas esta verificação apenas deve ser feita quando se executa o programa).

O formato de representação de uma expressão composta é como se segue:

```
( nome-do-operador argumento-1 ... argumento-N )
```

Alguns operadores têm um número fixo de argumentos, enquanto outros têm um número variável de argumentos (operadores variádicos).

Exemplos:

- **(add 1 2)** -- representa a operação **add** com dois argumentos (expressões **1** e **2**), tendo o significado **1+2**
- **(mul (add 1 2) 3)** -- representa a operação **mul** com dois argumentos (expressões **(add 1 2)** e **3**), tendo o significado **(1+2)*3**

- **(eq id1 id2)** -- representa a operação **eq** com dois argumentos (expressões designadas pelos identificadores **id1** e **id2**) e testa se os valores das expressões referidas pelos identificadores **id1** e **id2** são iguais

Tabela de operadores

Os vários operadores e a correspondente semântica são apresentados na tabela seguinte. A maioria dos operadores segue a semântica da família da linguagem C, excepto onde explicitamente indicado. Tal como em C, os valores lógicos são 0 (zero) (valor falso), e diferente de zero (valor verdadeiro). A ordem de avaliação dos argumentos é sempre da esquerda para a direita, excepto onde indicado (note-se que este comportamento não é uniforme na linguagem C).

Designação	Operadores	Operandos	Semântica
aritméticos (unários)	**neg**	inteiros	C
aritméticos (binários)	**add sub mul div mod**	inteiros	C
comparação (binários)	**lt le ge gt**	inteiros	C
igualdade (binários)	**eq ne**	inteiros	C
lógicos (unários)	**not**	inteiros	C
lógicos (binários)	**and or**	inteiros	C -- Para **and**, o 2º argumento só é avaliado se o 1º não for falso. Para **or**, o 2º argumento só é avaliado se o 1º não for verdadeiro.
atribuição (binário)	**set**	todos os tipos	O valor da expressão (2º argumento) é associado ao identificador (1º argumento). O valor da expressão é o do seu 2º argumento.
sequência (variádico)	**seq**	todos os tipos	As sub-expressões são avaliadas em sequência. O valor da expressão é o do seu último argumento.
impressão (variádico)	**print**	todos os tipos	As sub-expressões são avaliadas em sequência e os valores correspondentes apresentados na saída. O valor da expressão é o do seu último argumento.
leitura (sem argumentos)	**readi reads**	-	Pedem a introdução (pelo utilizador) de valores com os tipos correspondentes.

condicional (trenário)	**if**	condição (1º argumento) deve ser inteira	C -- Comporta-se de modo semelhante ao do operador trenário **?:** (não exige concordância de tipos entre os 2º e 3º argumentos). Apenas é avaliado o argumento correspondente ao valor lógico indicado pela condição.
ciclo (binário)	**while**	condição (1º argumento) deve ser inteira	Avalia o 1º argumento: se o valor lógico for falso, retorna-o; se o valor lógico for verdadeiro, avalia o 2º argumento, após o que reinicia o ciclo com nova avaliação do 1º argumento, etc. O valor da expressão é o valor da mais recente avaliação do seu primeiro argumento, ou seja, devolve sempre o valor correspondente à condição falsa (**0**, zero).
chamada (unário)	**call**	o nome do programa a chamar deve ser uma cadeia de caracteres literal	Executa o programa nomeado pelo argumento, devolvendo o valor da sua última expressão. Exemplo: **(call "nome-do-programa")**

Exemplos de expressões compostas

Os exemplos seguintes apresentam casos de uso simples dos operadores acima.

Exemplo 1: sequência com 3 expressões (valor **9**):

- **(seq (add 1 2) (sub 1 2) (mul (add 1 2) 3))**

Exemplo 2: sequência com 5 expressões (valor **"olá"**):

- **(seq (add 1 2) (sub 1 2) (mul (add 1 2) 3) (div 2 5) "olá")**

Exemplo 3: sequência com 2 expressões (valor final **0**) (note-se que esta sequência corresponde a um pequeno "programa" que apresenta o valor do identificador **ix** na saída):

- **(seq (set ix 0) (while (lt ix 30) (seq (print "ix =" ix) (set ix (add ix 1)))))**

Estrutura de um programa

Um programa é formado por um conjunto de expressões, tem um nome e pode ser executado. A execução de um programa corresponde a avaliar todas as suas expressões sequencialmente, com início na primeira.

Um programa pode não ter expressões (programa vazio). O valor de um programa vazio é **0** (zero).

Funcionalidade do Interpretador

O interpretador permite interpretar (representar o texto como objectos) e avaliar expressões (calcular os seus valores). Possui também várias formas de preservar o seu estado (não é possível manter várias versões do estado do interpretador em simultâneo).

Interpretação de expressões

As expressões, apresentadas ao interpretador em forma de texto, devem ser interpretadas antes de serem armazenadas. O texto original não é preservado. Note-se que o processo de interpretação (conversão do texto para estruturas/objectos) é diferente do de avaliação (conversão das estruturas/objectos para valores).

O texto a interpretar (expressões ou programas) pode ser providenciado via interface do próprio interpretador ou na forma de um ficheiro que contém um programa.

Uma falha de interpretação causa o lançamento de uma excepção no analisador do interpretador.

Note-se que não é necessário implementar de raiz o analisador de expressões e programas do interpretador. Será disponibilizado um analisador já parcialmente concretizado. Este analisador é responsável por processar um programa ou uma expressão e realizar a conversão de texto para a estrutura de dados correspondente. Será apenas necessário indicar as entidades utilizadas para representar um programa ou uma expressão.

Armazenamento de programas

O interpretador permite associar nomes a programas, preservando-os. Note-se que isto não corresponde a guardar valores calculados por esses programas, mas a guardar as suas descrições, i.e., as resultantes da interpretação da forma textual correspondente. Se um nome já estiver em uso, a associação ao programa anterior é perdida.

Interpretação de programas

A interpretação de programas e, consequentemente, das suas expressões, deve ser feita de forma flexível. Ou seja, deve ser possível -- sem alterar o código das expressões ou do interpretador -- definir novas formas de avaliação. Por omissão, a avaliação corresponde simplesmente à apresentação de uma forma textual do programa (e das suas expressões).

Uma outra avaliação possível é o cálculo dos valores das expressões do programa, correspondente à execução desse programa.

Serialização

É possível reiniciar, guardar e recuperar o estado actual do interpretador, preservando todos os programas e correspondentes nomes.

Interacção com o Utilizador

Descreve-se nesta secção a **funcionalidade máxima** da interface com o utilizador. Em geral, os comandos pedem toda a informação antes de proceder à sua validação (excepto onde indicado). Todos os menus têm automaticamente a opção **Sair** (fecha o menu).

As operações de pedido e apresentação de informação ao utilizador **devem** realizar-se através dos objectos *form* e *display*, respectivamente, presentes em cada comando. As mensagens são produzidas pelos métodos das bibliotecas de suporte (**po-uuilib** e **pex-app**). As mensagens não podem ser usadas no núcleo da aplicação (**pex-core**). Além disso, não podem ser definidas novas. Potenciais omissões devem ser esclarecidas antes de qualquer implementação.

As excepções usadas na interacção, excepto se indicado, são subclasses de **pt.tecnico.po.ui.DialogException**, são lançadas pelos comandos e tratadas por **pt.tecnico.po.ui.Menu**. Outras excepções não devem substituir as fornecidas nos casos descritos.

Note-se que os comandos e menus a seguir descritos, assim como o programa principal, já estão parcialmente implementados nas classes das *packages* **pex.app**, **pex.app.main** e **pex.app.evaluator**. Estas classes são de uso obrigatório e estão disponíveis no CVS (módulo **pex-app**).

Menu Principal

As acções do menu, listadas em **pex.app.main.MenuEntry**, permitem gerir a salvaguarda do estado da aplicação: Criar, Abrir, Guardar, Criar Programa, Ler Programa, Escrever Programa e Manipulação de Programa. A classe **pex.app.main.Message** define os métodos para geração das mensagens de diálogo. Inicialmente, o interpretador está vazio.

Salvaguarda do Estado Actual do Interpretador

O conteúdo do interpretador (inclui todos os programas actualmente carregados pelo interpretador) pode ser guardado para posterior recuperação (via serialização Java: **java.io.Serializable**). Na leitura e escrita do estado da aplicação, devem ser tratadas as excepções associadas. A funcionalidade é a seguinte:

- **Criar** -- Cria um novo interpretador sem ficheiro associado.
- **Abrir** -- Carrega um interpretador anteriormente salvaguardado, ficando o interpretador carregado associado ao ficheiro nomeado: pede-se o nome do ficheiro a abrir (**openFile()**). Caso o ficheiro não exista, é apresentada a mensagem **fileNotFound()**.
- **Guardar** -- Guarda o estado actual do interpretador no ficheiro associado. Se não existir associação, pede-se o nome do ficheiro a utilizar, ficando a ele associado. Esta interacção realiza-se através do método **newSaveAs()**. Não é executada nenhuma acção se não existirem alterações desde a última salvaguarda.

A opção **Sair** nunca guarda o estado da aplicação, mesmo que existam alterações.

Estes comandos já estão parcialmente implementados nas classes da *package* **pex.app.main** (disponível no CVS), respectivamente: **New**, **Open**, **Save**.

Criação, Leitura e Escrita de Programas

É possível criar novos programas (vazios), ler programas a partir de ficheiros textuais e escrever programas sob a forma de ficheiros textuais. As operações são as seguintes:

- **Criar Programa** -- Permite criar um programa vazio. É pedido o nome do programa através de **requestProgramId()**. A utilização de um nome previamente registado substitui o programa a ele associado por um programa vazio.
- **Ler Programa** -- Permite ler e interpretar o texto de um programa a partir de um ficheiro. É lido o ficheiro indicado pelo método **programFileName()**. A leitura de novo ficheiro com o mesmo nome substitui o programa anterior com esse nome (nome do ficheiro indicado).

- **Escrever Programa** -- Permite guardar um programa como um ficheiro de texto, passível de ser lido novamente pelo interpretador. O interpretador pede o identificador do programa **requestProgramId()** e o nome do ficheiro onde deve ser guardado o programa, através do método **programFileName()**. Escritas no mesmo ficheiro substituem o conteúdo anterior, não ficando associado ao interpretador.


Estes comandos já estão parcialmente implementados nas classes da *package* **pex.app.main** (disponível no CVS), respectivamente: **NewProgram**, **ReadProgram**, **WriteProgram**.

Manipulação de Programa

Abre o menu de edição de um programa e do seu conteúdo. O interpretador pede o identificador do programa **requestProgramId()**. Se o programa não existir, é comunicado o erro através de **noSuchProgram()**.


Este comando já está parcialmente implementado na classe **pex.app.main.EditProgram** (disponível no CVS).

Menu de Manipulação de Programas

Este menu permite efectuar operações sobre um programa. A lista completa é a seguinte: Listar programa, Executar, Adicionar expressão, Substituir expressão, Mostrar os identificadores presentes no programa, Mostrar os identificadores não inicializados do programa.

As etiquetas das opções deste menu estão definidas na classe **pex.app.evaluator.Label**. Todos os métodos correspondentes às mensagens de diálogo para este menu estão definidos na classe **pex.app.evaluator.Message**.


Estes comandos já estão parcialmente implementados nas classes da *package* **pex.app.evaluator** (disponível no CVS), respectivamente: **ShowProgram**, **RunProgram**, **AddExpression**, **ReplaceExpression**, **ShowAllIdentifiers**, **ShowUninitializedIdentifiers**.

Listar programa

Este comando apresenta a lista de expressões do programa em formato textual.

Executar programa

Este comando executa o programa.

Adicionar expressão ao programa

Este comando permite adicionar uma nova expressão ao programa. Para tal, é pedida a posição de inserção, através de **requestPosition()**, sendo a nova expressão aí inserida. A nova expressão é lida como resposta a **requestExpression()**.

Se a indicação de posição for inválida, o comando deve lançar a excepção **pex.app.BadPositionException** e o programa não é alterado.

Note-se que a nova expressão é interpretada: se a excepção **pex.ParserException** (do "core") for recebida, então a excepção **pex.app.BadExpressionException** deve ser lançada pelo comando. Neste caso, o programa não deve ser alterado.

Substituir expressão no programa

Este comando permite substituir uma expressão existente no programa por uma nova expressão. Para tal, é pedida a posição de inserção, através de **requestPosition()**, sendo a expressão aí existente substituída pela nova expressão. A nova expressão é lida como resposta a **requestExpression()**.

Se a indicação de posição for inválida, o comando deve lançar a excepção **pex.app.BadPositionException** e o programa não é alterado.

Note-se que a nova expressão é interpretada: se a excepção **pex.ParserException** (do "core") for recebida, então a excepção **pex.app.BadExpressionException** deve ser lançada pelo comando. Neste caso, o programa não deve ser alterado.

Mostrar os identificadores presentes no programa

Este comando permite listar todos os identificadores presentes no programa (apresentando um identificador por linha, por ordem alfabética), tanto nas expressões de definição, como nas que usam identificadores.

Mostrar os identificadores do programa sem inicialização explícita

Este comando permite listar todos os identificadores presentes no programa que não são objecto de nenhuma incialização explícita (apresentando um identificador por linha, por ordem alfabética), i.e., não terem um valor previamente associado via operador **set**.

Leitura de Programas a Partir de Ficheiros Textuais

Além das opções de manipulação de ficheiros descritas no menu principal, é possível iniciar a aplicação com um ficheiro de texto especificado pela propriedade Java **import**. Este ficheiro contém um programa que é avaliado pelo interpretador. O programa fica registado com o nome **import**.

Exemplo de ficheiro a importar

Este programa é composto por três expressões, a última das quais é um ciclo.

```
(seq (add 1 2) (sub 1 2) (mul (add 1 2) 3))
(seq (add 1 2) (sub 1 2) (mul (add 1 2) 3) (div 2 5) "olá")
(seq (set ix 0) (while (lt ix 30) (seq (print "ix =" ix) (set ix (add ix 1)))))
```

Exemplo de pequeno "programa"

Este exemplo corresponde a um pequeno "programa" (**triangulos.pex**) para verificar se três segmentos de recta formam um triângulo.

```
(print "Introduza as dimensões do 1º lado do triângulo: ")
(set a (readi))
(print "Introduza as dimensões do 2º lado do triângulo: ")
(set b (readi))
(print "Introduza as dimensões do 3º lado do triângulo: ")
(set c (readi))
(if (lt a 1)
    (print "As dimensões dos lados do triângulo devem ser positivas")
    (if (lt b 1)
        (print "As dimensões dos lados do triângulo devem ser positivas")
        (if (lt c 1)
            (print "As dimensões dos lados do triângulo devem ser positivas")
            (if (le (add a b) c)
                (print "Não é um triângulo")
                (if (le (add a c) b)
                    (print "Não é um triângulo")
                    (if (le (add c b) a)
                        (print "Não é um triângulo")
                        (if (eq a b)
                            (if (eq b c)
                                (print "Triângulo equilátero")
                                (print "Triângulo isósceles"))
                            (if (eq b c)
                                (print "Triângulo isósceles")
                                (print "Triângulo escaleno")))))))))
```

Embora o interpretador de expressões tenha de assinalar problemas relativos à interpretação de expressões mal especificadas, assume-se que não existem entradas mal-formadas nestes ficheiros (embora tenham de ser detectadas).

Considerações sobre Flexibilidade e Eficiência

Devem ser possíveis extensões ou alterações de funcionalidade com impacto mínimo no código já produzido para a aplicação. O objectivo é aumentar a flexibilidade da aplicação relativamente ao suporte de novas funções. Em particular, a solução encontrada para salvaguardar textualmente o conteúdo do documento deve ser suficientemente flexível de modo a permitir visualizar o conteúdo de um documento noutro formato (por exemplo, XML) sem que isso implique alterações no código *core* da aplicação.

Execução dos Programas e Testes Automáticos

Usando os ficheiros **test.pex**, **test.in** e **test.out**, é possível verificar automaticamente o resultado correcto do programa. Note-se que é necessária a definição apropriada da variável **CLASSPATH** (ou da opção equivalente **-cp** do comando **java**), para localizar as classes do programa, incluindo a que contém o método correspondente ao ponto de entrada da aplicação (**pex.app.App.main**). As propriedades são tratadas automaticamente pelo código de apoio.

```
      java -Dimport=test.pex -Din=test.in -Dout=test.outhyp pex.app.App
```

Assumindo que aqueles ficheiros estão no directório onde é dado o comando de execução, o programa produz o ficheiro de saída **test.outhyp**. Em caso de sucesso, os ficheiros das saídas esperada (**test.out**) e obtida (**test.outhyp**) devem ser iguais. A comparação pode ser feita com o comando:

```
diff -b test.out test.outhyp
```

Este comando não deve produzir qualquer resultado quando os ficheiros são iguais. Note-se, contudo, que este teste não garante o correcto funcionamento do código desenvolvido, apenas verificando alguns aspectos da sua funcionalidade.

Categories: **Ensino** **PO** **Projecto de PO**