TÉCNICO LISBOA

Projecto de Programação com Objectos

26 de Outubro de 2016

Esclarecimento de dúvidas:

- Consultar sempre o corpo docente atempadamente: presencialmente ou através do endereço **po-tagus@disciplinas.ist.utl.pt**.
- 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 na secção **[Projecto]** no Fénix):

- 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 nas secções **[Projecto]** e **[Método de Avaliação]** no Fénix):

- Datas: **2016/10/21 15:00** (UML); **2015/11/21 23:59** (intercalar); **2016/12/09 23:59** (final); **2016/12/19** (teste prático).
- Os diagramas de classe UML são entregues exclusivamente em papel (impressos ou manuscritos) na portaria do Tagus. Diagramas ilegíveis ou sem identificação completa serão sumariamente ignorados. É obrigatório a identificação do grupo e turno de laboratório (dia, hora e sala).
- **Apenas se consideram para avaliação os projetos submetidos no Fénix.** As classes criadas de acordo com as especificações fornecidas devem ser empacotadas num arquivo de nome `proj.jar` (que deverá apenas conter os ficheiros `.java` do código realizado guardados nos *packages* correctos). O ficheiro `proj.jar` deve ser entregue para avaliação através da ligação presente no Fénix. É possível realizar múltiplas entregas do projecto até à data limite, mas apenas será avaliada a última entrega.
- Não serão consideradas quaisquer alterações aos ficheiros de apoio disponibilizados: eventuais entregas dessas alterações serão automaticamente substituídas durante a avaliação da funcionalidade do código entregue.
- Trabalhos não presentes no Fénix no final do prazo têm classificação 0 (zero) (não são aceites outras formas de entrega).
- A avaliação do projecto pressupõe o compromisso de honra de que o trabalho 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 em 2016/2017**.

O objectivo deste projecto é concretizar uma aplicação que tem a funcionalidade de um interpretador. Este interpretador permite gerir um conjunto de programas e as expressões de que são compostos. O interpretador suporta uma linguagem de programação simples. A secção 1 apresenta as entidades do domínio da aplicação a desenvolver. As funcionalidades da aplicação a desenvolver são descritas nas secções 2, 3 e 4.

Neste texto, o tipo `fixo` indica um literal; o símbolo ␣ indica um espaço; e o tipo *itálico* indica uma parte variável.

1 Entidades do Domínio

Nesta secção descrevem-se as várias entidades que vão ser manipuladas no contexto da aplicação a desenvolver. O interpretador pode manter vários programas mas apenas executa um de cada vez. O nome de cada programa é único no contexto do interpretador.

1.1 Estrutura de um Programa

Um programa é formado por um conjunto de expressões. Um programa 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).

1.2 Estrutura de uma Expressão

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

Genericamente, 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, sobre a qual pode ser então executado o processo de avaliação, para determinação do valor correspondente. No contexto deste projecto quer-se apenas considerar a segunda abordagem.

A linguagem utilizada para descrever um programa e que deve ser suportada aplicação a desenvolver é uma linguagem simples e uniforme. Nesta linguagem, uma expressão pode ser categorizada como *expressão primitiva* (literais e identificadores) ou como *expressão composta* (operadores).

1.2.1 Expressões Primitivas

Há dois tipos de *expressão primitiva*. Uma expressão primitiva pode representar directamente um valor (designada como *expressão literal*), ou pode referir outra expressão (designada como *identificador*).

No contexto da linguagem simplificada a suportar, um identificador na realidade representa apenas uma variável. Numa linguagem mais poderosa, um identificador poderia representar outras tipos de entidades, por exemplo, um método ou uma classe.

Literais Existem dois tipos de expressões literais: inteiros e cadeias de caracteres.

Um literal inteiro é um número inteiro não negativo e é constituído por uma sequência de 1 (um) ou mais dígitos de 0 a 9. Exemplos: 1 e 23.

As cadeias de caracteres são delimitadas por aspas (") e podem conter quaisquer caracteres. Exemplos: "hello world!"e "abcd".

Identificadores Um identificador é representado por um nome que é uma sequência de caracteres com a seguinte restrição: a sequência é composta por um ou mais caracteres alfanuméricos e o primeiro caracter tem que ser uma letra. O tamanho da sequência é ilimitado e dois nomes são iguais se forem constituídos pela mesma sequência de caracteres.

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

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

Os identificadores definidos num programa são visíveis em todos os programas, ou seja, os identificadores são *partilhados* por todos os programas de um interpretador.

1.2.2 Expressões Compostas

Uma expressão composta é constituída por um operador e zero, um ou mais argumentos. Cada argumento é representado por uma expressão. Alguns operadores têm um número fixo de argumentos (0, um ou dois), enquanto outros têm um número variável de argumentos (designados como operadores variádicos).

O valor de uma expressão composta corresponde à avaliação do seu 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, há operadores que só suportam argumentos que devolvem um número 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)
```

Exemplos:

- **(add 1 2**) – representa a operação **add** com dois argumentos (expressões literais **1** e **2**), tendo o significado Java **1+2**
- **(mul (add 1 2) 3)** – representa a operação **mu**l com dois argumentos (expressão composta **(add 1 2)** e expressão literal **3**), tendo o significado Java **(1+2)*3**
- **(eq id1 id2)** – representa a operação **eq** com dois argumentos (expressões identificadores designadas pelos identificadores **id1** e **id2)** e verifica se os valores das expressões referidas pelos identificadores id1 e id2 são iguais

Os vários operadores a suportar pela linguagem e a correspondente semântica são apresentados na tabela 1. A maioria dos operadores segue a semântica da família da linguagem C, excepto onde seja explicitamente indicado. Tal como em C, os valores lógicos são representados por 0 (zero) (valor falso), e diferente de zero (valor verdadeiro).

Designação	Operador	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 (inteiro para `readi` e cadeia de caracteres para `reads`).
condicional (trenário)	if	1º argumento deve devolver um inteiro	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	1º argumento deve devolver um inteiro	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 (unitário)	call	o nome do programa a chamar deve ser uma cadeia de caracteres	Executa o programa nomeado pelo argumento, devolvendo o valor da sua última expressão. Exemplo: **(call "nome-do-programa")**

Tabela 1: Tabela com a descrição da semântica e sintáxe dos operadores da linguagem.

A expressão composta *print* muda de linha após escrever o último argumento (ou seja, adiciona automaticamente um `\n`).

Exemplos de expressões compostas: Os exemplos seguintes apresentam casos de uso simples dos operadores apresentados na tabela 1:

- Sequência com 3 expressões (valor 9): **(seq (add 1 2) (sub 1 2) (mul (add 1 2) 3))**
- Sequência com 5 expressões (valor "olá"): **(seq (add 1 2) (sub 1 2) (mul (add 1 2) 32) (div 2 0) "olá")**
- Sequência com 2 expressões (valor 0): **(seq (set ix 0) (while (lt ix 30) (seq (print "ix =" ix) (set ix (add ix 1)))))**. Note-se que esta sequência corresponde a um pequeno "programa"que vai apresentando ao utilizador os diferentes valores atribuídos ao identificador `ix` durante a avaliação desta expressão. O valor devolvido por esta expressão é 0 (zero), tal como descrito na tabela 1 relativamente ao comportamento do operador `while`

2 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.

2.1 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.

Note-se que os alunos não têm que concretizar 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. Os alunos terão apenas que indicar as entidades que utilizam para representar um programa ou uma expressão.

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

2.2 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.

Deve ser possível reiniciar, guardar e recuperar o estado actual do interpretador (utilizando o mecanismo de *serialização* do Java), preservando todos os programas e correspondentes nomes.

2.3 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 interpretação corresponde simplesmente à apresentação de uma forma textual do programa (e das suas expressões).

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

3 Interação com o Utilizador

Descreve-se abaixo 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 leitura e escrita **devem** realizar-se exclusivamente através de duas classes: `pt.utl.ist.po.ui.Form` e `pt.utl.ist.po.ui.Display`. As mensagens a apresentar durante a execução da aplicação são produzidas pelos métodos das bibliotecas de suporte (**po-uilib** e **pex-support**). Não deve haver código de interacção com o utilizador no núcleo da aplicação (*core'*). Desta forma,será possível reutilizar o código do núcleo da aplicação com outras concretizações da interface com o utilizador. Além disso, não podem ser definidas novas mensagens. Potenciais omissões devem ser esclarecidas antes de qualquer concretização.

As várias situações de erro que podem acontecer durante a execução da aplicação a desenvolver devem ser representadas por excepções distintas que são subclasses de `pt.utl.ist.po.ui.InvalidOperation` . Estas excepções devem ser lançadas pelos comandos por forma a poderem ser tratadas automaticamente pela *framework* de interacção com o utilizador **po-uilib**: A classe `pt.utl.ist.po.ui.Menu` já trata automaticamente as excepções que ocorram durante a execução dos comandos.

Note-se que os comandos e menus a seguir descritos, assim como o programa principal, já estão parcialmente implementados nas classes dos packages `pex.app`, `pex.app.main` e `pex.app.evaluator`. Estas classes são de uso obrigatório e estão disponíveis na secção *Projecto* da página da cadeira.

3.1 Menu Principal

As acções do menu, listadas em `pex.app.main.MenuEntry`, permitem gerir a salvaguarda do estado da aplicação (§3.1.1), assim como realizar operações sobre o interpretador actual: **Criar**, **Abrir**, **Guardar**, **Criar Programa**, **Ler Programa** , **Escrever Programa** e **Manipulação de Programas** (§3.2). A classe `pex.app.main.Message` define os métodos para geração das mensagens de diálogo.

Inicialmente, a aplicação não tem nenhum programa. A opção `Sair` **nunca** guarda o estado da aplicação, mesmo que existam alterações.

3.1.1 Salvaguarda do Estado Actual

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 (*anónimo*) não associado a nenhum ficheiro.

Abrir – Carrega um interpretador anteriormente salvaguardado, ficando o interpretador carregado associado ao ficheiro nomeado. A cadeia de caracteres a utilizar para pedir o nome do ficheiro a abrir é a devolvida pelo método `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 associada. Esta interacção realiza-se através do método `newSaveAs()`. Não é executada nenhuma acção se não existirem alterações desde a última salvaguarda.

Note-se que não é possível manter várias versões do estado do interpretador em simultâneo.

Estes comandos já estão parcialmente implementados nas classes do package **pex.app.main**: **New**, **Open** e **Save** .

3.1.2 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** – Cria 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** –Lê e interpreta 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** – Guarda 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()`. Se o programa não existir, é comunicado o erro através de `noSuchProgram()`. Escritas no mesmo ficheiro substituem o conteúdo anterior, não ficando associado ao interpretador.

Estes comandos já estão parcialmente implementados nas classes do package **pex.app.main**: **NewProgram**, **ReadProgram** e **WriteProgram**.

3.1.3 Manipulação de Programas

Abre o menu de edição (alteração) 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`.

3.2 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** e **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 concretizados nas classes já disponibilizadas do *package* `pex.app.evaluator`. **ShowProgram**, **RunProgram**, **AddExpression**, **ReplaceExpression**, **ShowAllIdentifiers** e **ShowUninitializedIdentifiers**.

3.2.1 Listar programa

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

3.2.2 Executar

Este comando executa o programa.

3.2.3 Adicionar Expressão ao Programa

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

Se a indicação de posição for igual ao número de elementos do programa, a nova expressão é inserida no final do programa (após todas as outras). 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 deve ser processada pelo analisador de expressões. Se este lançar a excepção `pex.ParserException` (do "core"), então a excepção `pex.app.BadExpressionException` deve ser lançada pelo comando e o programa não deve ser alterado.

3.2.4 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 deve ser processada pelo analisador de expressões. Se este lançar a excepção `pex.ParserException` (do "core"), então a excepção `pex.app.BadExpressionException` deve ser lançada pelo comando e o programa não deve ser alterado.

3.2.5 Mostrar os identificadores presentes no programa

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

3.2.6 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 (apresentado um identificador por linha, por ordem alfabética), i.e., não terem um valor associado via operador **set**.

4 Leitura de Expressões a Partir de Ficheiros Textuais

Além das opções de manipulação de ficheiros descritas em §3.1.1, é possível iniciar a aplicação com um ficheiro de texto especificado pela propriedade Java **import**. Este ficheiro contém a descrição de um programa. O programa fica registado com o nome **import**. Quando se especifica esta propriedade, é criado um interpretador já com um programa com o nome indicado e conteúdo correspondente ao presente no ficheiro fornecido.

4.1 Exemplo de Pequeno Programa

Este exemplo corresponde a um pequeno programa para verificar se três segmentos de recta formam um triângulo válido.

```
(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")))))))))
```

5 Considerações sobre Flexibilidade e Eficiência

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

6 Execução dos Programas e Testes Automáticos

Usando os ficheiros `test.import`, `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 verifica alguns aspectos da sua funcionalidade.