1 Introdução

Como viram no segundo trabalho individual, a classe `StdDraw` disponibiliza uma biblioteca gráfica para imagens 2D. Neste projeto vamos criar uma DSL (*domain specific language*) para manipular a classe `StdDraw` sem ter de compilar programas Java. Assim, se quisermos criar uma imagem 2D, bastará escrever um *script* nessa DSL, e executá-lo, por exemplo, na linha de comando.

Para tal é necessário criar um programa Java capaz de ler um ficheiro de texto com um algoritmo descrito na DSL, que execute este algoritmo, e que nos devolva o resultado. À linguagem vamos chamar AEDLANG.

A AEDLANG pode receber dados pelo *input stream* (normalmente da consola ou de um ficheiro de texto). A AEDLANG pode escrever tanto para o *output stream* como para a janela gráfica do `StdDraw`.

A DSL que iremos conceber é um exemplo de uma *linguagem concatenativa*.

Numa linguagem de programação concatenativa todos os elementos de uma expressão são funções, e uma sequência de funções separadas por espaços corresponde à composição dessas funções. Por exemplo, a expressão `x f g` representa o resultado da composição `g(f(x))`. Note que se usa uma notação sufixa, e portanto os argumentos surgem antes da função, sendo que a última função a aplicar é dada mais à direita.

Outro exemplo: o resultado das seguintes instruções Java:

```
x = f(w);
y = g(x);
h(y);
```

seria, numa linguagem concatenativa, descrito por:

```
w f g h
```

Valores são igualmente funções e surgem no mesmo formato:

```
5 2 * -1 +
```

Esta expressão resulta em 9 dado que representa $5 \times 2 + (-1)$. Note que as operações $*$ e $+$ requerem dois argumentos e por essa razão são precisos dois valores antes de cada uma das funções: 5 e 2 para a multiplicação; o resultado da multiplicação e -1 para a soma.

2 Programas

Um programa AEDLANG é uma sequência de linhas de texto. Estas linhas de texto podem ser de três tipos:

1. comentários (linhas começadas por um #), não há comentários a meio das linhas
2. definições de função
3. expressões AEDLANG que são a composição de múltiplas funções

Um exemplo inicial de um programa AEDLANG (veremos adiante os detalhes sintáticos):

```
# a função unaria que calcula o quadrado do argumento
# $1 é o nome do argumento da função
[quadrado:1] $1 $1 *

# uma expressao que resulta em (3.5+1)^2
3.5 1 + quadrado
```

3 Execução e Estado

Vamos começar por explicar como se executa uma expressão da linguagem AEDLANG.

As funções, como vimos na introdução, apresentam-se no formato sufixo, ou seja, surgem depois dos seus argumentos. Este formato sufixo pode ser facilmente processado recorrendo ao uso de uma pilha de dados.

A avaliação de uma expressão sufixa usando uma pilha é feita do seguinte modo:

- consome-se elemento a elemento da expressão (designados por **tokens**). Os *tokens* têm de ser separados, entre si, por um ou mais espaços.
- cada *token* consumido produz um efeito. Por exemplo, pode incluir um elemento no topo da pilha (como no caso dos números) ou, se for um operador, pode remover um número suficiente de argumentos, avaliar o operador nesses argumentos e colocar o resultado no topo da pilha.

Vejamos um exemplo com a expressão `5 2 * -1 +`:

Expressão a processar	Pilha
`5 2 * -1 +`	↔]
`2 * -1 +`	↔ 5]
`* -1 +`	↔ 2,5]
`-1 +`	↔ 10]
`+`	↔ -1,10]
`<vazio>`	↔ 9]

No fim da execução, o resultado está no topo da pilha.

O *estado da pilha* determina o resultado da próxima função. O estado da pilha é um tuplo com as seguintes componentes:

- o conteúdo da pilha, i.e., os valores que ela armazena
- as coordenadas `x,y` atuais, i.e., onde estamos na janela gráfica do `StdDraw`
- se é para preencher ou não as figuras geométricas (se é ou não é para fazer *fill* nos comandos do `StdDraw`).

A execução de um programa começa no estado inicial (a pilha está vazia), e vai mudando de estado em estado de acordo com as funções executadas. A tabela anterior mostra-nos isso. A coluna da direita mostra o estado (no que toca a esta pilha em particular), sendo que a execução da expressão produziu seis estados desde o estado inicial (onde a pilha estava vazia) até ao estado final (onde a pilha contém o número 9).

No que consiste o estado de um programa? *O estado de um programa é definido por uma lista de estados de pilhas.*

Vamos considerar que a AEDLANG apenas admite o tipo de dados `Double`. Assim, as pilhas de dados que iremos manipular serão pilhas de *doubles*.

4 A linguagem

4.1 Funções pré-definidas

Existe um conjunto de funções que devem existir *a priori*.

A linguagem deve incluir um conjunto de *tokens* que correspondem aos operadores aritméticos tradicionais para números. Estes são o $+, *, /, \hat{}$ (potência). Para simplificar, não iremos usar a subtração e reservamos o símbolo '-' para os números negativos.

Como não há tipo booleano, assumimos que `False` é zero, e `True` é diferente de zero. Os *tokens* relativos a operações booleanas são similares aos do Java:

- operadores relacionais: `< > =`
- operadores booleanos: `! & |`

Devemos ter os *tokens* `sin` e `cos` que calculam o seno e o coseno respetivamente, e `rnd` que devolve um número aleatório entre 0.0 e 1.0.

A seguir temos a lista de funções para manipular pilhas:

- `dup`, que empilha novamente o topo da pilha (eg, ↔ `1,2]` fica ↔ `1,1,2]`)
- `swap`: troca a posição dos dois elementos mais acima na pilha (eg, ↔ `1,2,3]` fica ↔ `2,1,3]`)
- `pop`: retira o topo da pilha (eg, ↔ `2,1]` fica ↔ `1]`)
- `size`: coloca no topo da pilha o número de elementos da pilha (eg, ↔ `5,3,7,7]` fica ↔ `4,5,3,7,7]`)
- `eos`: coloca 1.0 na pilha se esta estava vazia, 0.0 caso contrário
- `nil`: não faz nada

Há duas funções para manipular a lista de pilhas:

- `next`: passamos a manipular a pilha seguinte da lista (se não existir, deve-se criar uma nova pilha vazia).
- `prev`: passamos a manipular a pilha anterior da lista (se não existir, deve-se criar uma nova pilha vazia).

A seguinte lista refere funções que produzem *output* na janela gráfica:

- `window`, produz uma janela gráfica (cf detalhes na secção 6.1)
- `point`, recolhe `x` e `y` da pilha, e desenha esse ponto na janela gráfica
- `move`, recolhe `x` e `y` da pilha, e posiciona o estado da pilha atual em (`x`, `y`)
- `line`, recolhe `x` e `y` da pilha, desenha uma linha entre o ponto do estado atual e o ponto (`x`, `y`), e posiciona o ponto do estado atual em (`x`, `y`)
- `rline`, recolhe Δx e Δy da pilha, desenha uma linha do estado atual (x_0, y_0) até $(x_0 + \Delta x, y_0 + \Delta y)$, e posiciona o ponto do estado atual em $(x_0 + \Delta x, y_0 + \Delta y)$
- `square`, recolhe `r` da pilha, e desenha o quadrado de lado `2r` centrado no ponto do estado atual
- `rectangle`, recolhe `h,w` da pilha, e desenha o rectangulo de dimensões `2h,2w` centrado no ponto do estado atual
- `circle`, recolhe `r` da pilha, e desenha o círculo de raio `r` centrado no ponto do estado atual
- `fill-on`, coloca o estado atual com preenchimento das figuras
- `fill-off`, coloca o estado atual sem preenchimento das figuras
- `rgb`, recolhe `r,g,b` da pilha, e define a cor atual da janela gráfica como sendo o triplo (r,g,b)

4.2 Definição de Funções

Uma funcionalidade que queremos é a possibilidade de definir novas funções (também designadas *macros*).

Uma função é composta por três elementos: o seu *nome*, a sua *aridade* (ie, quantos argumentos possui) e uma *expressão* a que corresponde o seu comportamento. Avaliar o *token* de uma função corresponde a avaliar a expressão associada a essa função.

Vejamos um exemplo. A seguinte função designada `dobro` tem zero argumentos (já iremos falar dos argumentos) e deve calcular o dobro do topo da pilha:

```
[dobro:0] 2 *
```

Com esta função definida podemos avaliar[1] a expressão `4 3 dobro +`.

Expressão a processar	Pilha
`4 3 dobro +`	↔]
`3 dobro +`	↔ 4]
`dobro +`	↔ 3,4]
`2 * +`	↔ 3,4]
`* +`	↔ 2,3,4]
`+`	↔ 6,4]
`<vazio>`	↔ 10]

Qual é o papel da aridade, ou seja, do número de argumentos de uma função? Uma função com aridade n retira n valores do topo da pilha, antes de avaliar a expressão associada. Estes elementos retirados serão os argumentos da função. A forma como acedemos aos seus valores posteriormente é usando a sintaxe `$1`, para o primeiro argumento, `$2`, para o segundo, e assim sucessivamente...

Podemos redefinir a aridade da função dobro para ter um argumento:

```
[dobro:1] 2 $1 *
```

Ou seja, agora a função, antes de ser avaliada, retira o topo da pilha e coloca-o no argumento `$1`. Vejamos como se desenrola a avaliação desta versão do dobro:

Expressão a processar	Pilha
`4 3 dobro +`	↔]
`3 dobro +`	↔ 4]
`dobro +`	↔ 3,4]
`2 $1 * +`	↔ 4]
`$1 * +`	↔ 2,4]
`* +`	↔ 3,2,4]
`+`	↔ 6,4]
`<vazio>`	↔ 10]

De notar que os vários `$i` são locais. Se for avaliada uma segunda função durante a avaliação da primeira, esta segunda função terá os seus próprios argumentos `$i`. Por exemplo, o seguinte programa deixaria 140 no topo da pilha atual. Executem este programa passo a passo para confirmar o resultado final de 140.

```
[f:2] $1 $2 g +
[g:1] $1 20 *

100 2 f
```

Devem também incluir o operador `$$` que, dentro da definição da função, empilha todos os argumentos da mesma, desde o `$1` até ao último argumento. O seguinte programa

[1] Esta tabela é só um exemplo. A forma como avaliam as várias expressões e gerem os resultados obtidos, é algo que vocês terão de pensar como pode ser implementado.

```
# funcao recolhe 3 numeros e empilha-os duas vezes seguidas
[h:3] $$ $$

1 2 3 h
```

deixaria a pilha no estado final ↔ 3 2 1 3 2 1]

4.3 Variáveis

A linguagem AEDLANG possui variáveis que guardam valores *double*. Cada variável tem um nome. Quando avaliamos esse nome, o programa deve colocar o conteúdo da variável no topo da pilha.

Para criar uma nova variável escrevemos `@nome`. Neste caso cria-se a nova variável com esse nome e inicializa-se com o topo da pilha, consumindo-o da pilha. Exemplo:

Expressão a processar	Pilha
`100 @xpto 5 xpto *`	↔]
`@xpto 5 xpto *`	↔ 100]
`5 xpto *`	↔]
`xpto *`	↔ 5]
`*`	↔ 100,5]
`<vazio>`	↔ 500]

A nossa sugestão é olharem para as variáveis como funções constantes de aridade zero, que devolvem o valor associado à variável. Assim, a expressão `100 @xpto` poderia ser avaliada como `[xpto:0] 100`.

Tanto as variáveis como as funções definidas no programa têm um contexto global. Não fazem parte do estado da pilha atual. Por exemplo, após executar

```
100 @x x next x
```

ambas as pilhas teriam no topo, o valor 100.

Não se pode usar os nomes das funções pré-definidas, seja nas vossas funções seja nas variáveis.

4.4 Input e Output

A função `?` lê uma expressão do *input stream* (seja uma linha de um ficheiro ou da consola) e avalia essa expressão. Por exemplo, se o utilizador introduzir um número, esse deverá ficar no topo da pilha atual.

Outra funcionalidade da AEDLANG é a capacidade de gerar *output* de texto. Para tal temos dois operadores, o `.` (ponto) e a `,` (vírgula). Ambos consomem o topo da pilha e enviam esse valor para o *output stream*. A única diferença é que o operador ponto adiciona uma mudança de linha e o operador vírgula adiciona um espaço. Exemplo:

Expressão a processar	Pilha	Output
`1 , 2 .`	↔]	
`, 2 .`	↔ 1]	
`2 .`	↔]	`"1 "`
`.`	↔ 2]	`"1 "`
`<vazio>`	↔]	`"1 2\n"`

4.5 Condicionais e Ciclos

Como não podia deixar de ser, a linguagem AEDLANG tem expressões condicionais e ciclos. Para facilitar a leitura destes *tokens* vamos considerar que só podem ocupar uma linha de texto.

Sintaxe do *if*:

```
{ bloco-expressão-then } { bloco-expressão-else } if
```

Reparem que tem de haver espaços a separar as chavetas das expressões internas (são os espaços que permitem separar os diferentes *tokens*).

O comportamento da função *if* é o seguinte: 1) retira o topo da pilha, 2) se esse valor é diferente de zero executa a expressão do bloco *then*, caso contrário executa a expressão do bloco *else*.

Exemplo: a seguinte função calcula o fatorial recursivamente, usando o if para determinar se já chegámos à base da recursão ($0! = 1$):

```
[fact:1] $1 0 = { 1 } { $1 -1 + fact $1 * } if
```

O que é suposto fazer quando o *token* seguinte é a abertura de uma chaveta? Esta e outras expressões condicionais não são para serem inseridas nas pilhas. Afinal, as pilhas apenas guardam *doubles*. Têm de pensar numa estrutura de dados auxiliar onde possam guardar as expressões que estão dentro das chavetas. Depois, consoante terem de executar o bloco *then* ou o bloco *else*, vão buscar a respetiva expressão e mandam avaliar na pilha atual, como qualquer outra expressão.

Para iterações a AEDLANG tem a *função* `loop`. A sua sintaxe:

```
{ bloco-expressão } loop
```

O comportamento do `loop` é o seguinte:

1. retira o topo da pilha (seja n),
2. repete para $i = 1 \ldots n$:
   (a) coloca i no topo da pilha e executa a expressão do bloco associado.

Se o número n lido for menor que 1, o ciclo não faz nada.

Exemplos:

- `5 { } loop` resulta na pilha ↔ `5,4,3,2,1]`
- `3 { dup } loop` resulta ↔ `3,3,2,2,1,1]`
- `1 6 { * } loop` resulta ↔ `720]`, porque calculou o fatorial de 6.
- `4 { 2 < { 0 } { 1 } if } loop` resulta ↔ `1,1,1,0]`

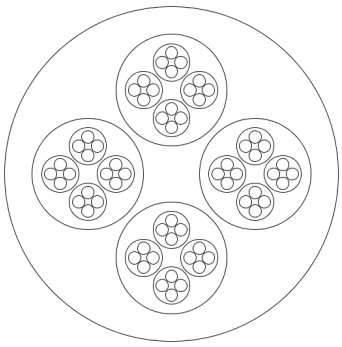
Como se pode ver neste último exemplo, pode haver blocos dentro de blocos.

5 Exemplos

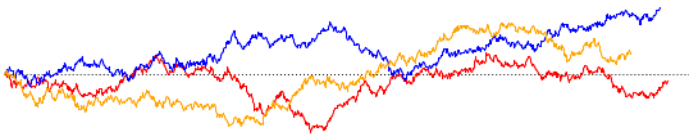
Seguem-se alguns exemplos de programas e o seu *output* esperado.

```
# draw a target
512 512 window
fill-on

[RED :0] 255 0 0 rgb
[BLACK:0] 0 0 0 rgb

[toggle:0] flip 0 = { 1 } { 0 } if @flip
[color:0] flip 0 = { RED } { BLACK } if

0 @flip

# must draw from larger to smaller, since these are filled circles
10 { 10 swap -1 * + 20 * @radius toggle color radius circle } loop
```

```
# drawing Koch circles
512 @win-size
win-size 1.1 * dup window

# params(x0,y0,size,iter)
[north:4] $1              $2 $3 -0.5 * +   $3 3 /    $4 -1 +
[south:4] $1              $2 $3  0.5 * +   $3 3 /    $4 -1 +
[east :4] $1 $3  0.5 * +  $2               $3 3 /    $4 -1 +
[west :4] $1 $3 -0.5 * +  $2               $3 3 /    $4 -1 +

[koch:4] $4 0 = { nil } { $1 $2 move $3 circle $$ north koch
$$ south koch $$ east koch $$ west koch } if

0 0 win-size 2 / 4 koch
```



```
# Random Walks
[RED :0]   255   0   0 rgb
[ORANGE:0] 255 165   0 rgb
[BLUE :0]    0   0 255 rgb

512 @win-size
win-size 1.1 * dup window

[rnd-walk:0] win-size -0.5 * 0 move  win-size 2 * { rnd rnd 5 * -2.5 + rline } loop

# draw horizontal line
win-size { 3 * win-size -2 / + 0 point } loop

RED    rnd-walk
BLUE   rnd-walk
ORANGE rnd-walk
```



6 O que fazer

Abram o projeto dado no Eclipse. É-vos pedido que implementem a classe `Executor`. Respeitem as assinaturas das classes fornecidas. Podem, naturalmente, incluir os métodos auxiliares que acharem adequados.

Incluam comentários informativos e *javadoc* dos vários métodos.

6.1 A interface estratégia

No pacote `strategies` têm esta interface disponível:

```
1 public interface Strategy {
2   public void execute(State state);
3 }
```

Qual a utilidade desta interface? Muitas das funções pré-definidas precisam apenas do estado da pilha atual para trabalharem. Criar uma classe estratégia por comando torna o código mais organizado e de mais fácil gestão.

Vejamos um exemplo de uso:

```
1  public class Swap implements Strategy {
2
3    public void execute(State state) {
4      double a = state.pop();
5      double b = state.pop();
6      state.push( a );
7      state.push( b );
8    }
9  }
```

Esta classe fica responsável por executar o comando `swap`. Basta desempilhar os dois elementos da pilha atual, e voltar a empilhar pela ordem trocada. Todo o código relativo ao respetivo comando está concentrado nesta classe.

Outro exemplo seria o comando `window`:

```
1  public class Window implements Strategy {
2
3    public void execute(State state) {
4      int ysize = (int) Math.floor(state.pop());
5      int xsize = (int) Math.floor(state.pop());
6
7      StdDraw.clear(StdDraw.WHITE);
8      StdDraw.setPenColor(StdDraw.BLACK);
9
10     StdDraw.setCanvasSize(xsize, ysize);
11     StdDraw.setXscale(-xsize/2, xsize/2);
12     StdDraw.setYscale(-ysize/2, ysize/2);
13   }
14 }
```

De notar que nem todas as funções podem ser implementadas assim. Por exemplo, as funções `next` e `prev` têm como tarefa *mudar* a pilha atual. Outros exemplos são o `loop` e o `if` que precisam de ter acesso aos blocos `{...}` que não fazem parte do estado da pilha atual.

Na classe `Executor`, organizem estes objetos `Strategy` numa classe que implemente `java.util.Map` (podem usar uma das classes das bibliotecas Java).

7 Entrega

Devem entregar um zip chamado `projetoAED_XXX`, sendo `XXX` o número de grupo. O zip deve conter o vosso projeto Eclipse. O zip deve ser entregue via moodle até às 23:59 do dia 20 de Abril.

Identifiquem o número do grupo e os vossos números de aluno e nomes no *javadoc* da classe.