Projeto de Laboratório de Programação II - Fase 2

Karina Suemi, Vinícius Silva, Renato Cordeiro

1 Introdução

Nesta fase, o projeto consiste na criação de uma máquina virtual em JAVA, juntamente com a implementação de funções que darão ao usuário o poder de controlar o robô através de comandos usados com pilha de dados. Há também a criação da Arena, esta é resposável por controlar os robôs de forma que eles sejam impossibilitados de fazer movimentos não propícios.

E por último, temos os tipos de terrenos e objetos que estarão dispostos nele, como cristais, rochas (como objetos), bases, água, árvores, entre outros (como cenário).

1.1 Projeto

Os arquivos estão separados de acordo comcolor a seguinte estrutura de diretórios:

- **bin:** Arquivo executável principal do programa;
- **doc:** Pasta para a documentação, com este arquivo e a versão fonte em LaTeX;
- **lib:** Arquivos com classes/pacotes de Perl no formato `.pm`;
- **test:** Arquivos de teste com código em Assembly do montador/máquina virtual `.txt`.

1.2 Estrutura

Para informações sobre o uso de programa e os seus desenvolvedores, consultar os arquivos `README.md` e `LICENSE`.

O programa consiste de três partes principais:

- Montador;
- Máquina Virtual;
- Emulador.

1.3 Emulador

O emulador é a parte mais simples do programa, e consiste de um arquivo do tipo `.pl` nomeado `robots.pl` na pasta `bin/`. Para executá-lo, basta rodar o programa passando como parâmetro na linha de comando um arquivo (de extensão qualquer) com o assembly válido.

Caso haja erro de compilação, o programa será encerrado antes da execução de qualquer comando. Caso o erro seja lógido (*runtime*), os erros serão impressos na saída de erros (STDERR) no próprio prompt de comando.

2 Montador

O montador tem como função ler o programa em texto (que será passado pelo usuário), ver se ele está formatado de maneira correta e em caso positivo, é criada uma matriz 3 x n, sendo n o número de comandos, que será passada para a Máquina Virtual.

O arquivo com o montador está no diretório de bibliotecas `lib/` e é chamado `Cortex.pm`

No caso, as linhas do programa de entrada terão que seguir o seguinte padrão:

[Comando, Argumento, Label]

O montador devolverá erro em caso de comandos que não sejam compatíveis com seus respectivos tipos de argumentos. No caso da entrada ser:

x **"ADD Olá"** O comando ADD não deveria receber argumentos, e como não é o caso, o programa passará *undef* para a Máquina Virtual.

✓ **"ADD"** Este seria o correto.

x **"PUSH Oie!"** O comando PUSH deveria receber argumentos numéricos, e como não é o caso, o programa passará *undef* para a Máquina Virtual.

✓ **"PUSH 100"** Este seria o correto.

x **"JIF 5uvas"** O comando JIF deveria receber argumentos em forma de texto (pode conter números, desde que não sejam 1º caracter), e como não é o caso, o programa passará *undef* para a Máquina Virtual.

✓ **"JIF uvas5"** Este seria o correto.

Segue a tabela com os argumentos necessários para cada tipo de comando:

Vazio	ADD - DIV - DUP - END - EQ - GE - GT MOD - LE - LT - MUL - NE - POP - PRN SUB - RET - MOVE - DRAG - DROP - HIT LOOK - ITEM - SEE - SEEK
Numérico	RCL - STO
String	JMP - JIF - JIT - CALL
Empilhável	crystal - stone

3 Máquina Virtual

A máquina virtual (RVM, *Robot Virtual Machine*, consiste em um conjunto de módulos localizados no diretório lib/RVM que integram o pacote/classe RVM.

Os arquivos do módulo são:

- RVM.pm
- Overload.pm
- ctrl.pm
- functions.pm
- f_arit.pm
- f_io.pm
- f_jumps.pm
- f_memo.pm
- f_stk.pm
- f_tests.pm

3.1 RVM.pm

A *Robot Virtual Machine* (**RVM**) é o pacote principal que controla o 'cérebro' dos robôs, servindo como o interpretador para o formato criado pelo montador. Cada instância da classe RVM possui os seguintes atributos:

- PROG: Vetor com os comandos do programa, conforme o formato criado pelo montador;
- DATA: Vetor com a pilha principal de memória, no qual serão armazenados os dados colocados com comandos de manipulação da pilha;
- PC: Registrador para a posição atual de execução dos comandos dentro do vetor PROG;
- RAM: Memória auxiliar, a ser utilizada para funções;
- CTRL: Vetor de controle dos retornos das funções (callbacks);
- LABEL: Hash com as posições associadas a cada LABEL criado como marcador no código da RVM.

O módulo RVM contém dois métodos, responsáveis por agirem como construtor (**new**) e inicializador de programa (**upload**) para os objetos da RVM.

3.2 Overload.pm

Módulo auxiliar que faz parte da classe da RVM. Sua principal função é fornecer a sobrecarga do operador *double quote* (**qq**// ou **"**), de modo a possibilitar a impressão dos valores armazenados dentro da RVM.

3.3 ctrl.pm

O módulo auxiliar do controlador contém o método responsável por realizar a execução do programa armazenado no atributo **PROG** do objeto da classe RVM.

Começando da posição 0, a cada posição de PROG realiza a chamada da função correspondente, usando os argumentos (se disponíveis ou necessários). Nesse processo, atualiza o PC (registrador de posição) para realizar a atualização necessária (avançar para o próximo comando ou fazer um desvio).

Quando a função retorna um erro, lança uma exceção (erro de *runtime*), que pode ser de 3 tipos:

- Falha de segmentação: tentativa de acessar região não definida da memória - overflow/underflow da pilha de dados DATA;
- Operação inválida: quando o operando não é do tipo válido;
- Label desconhecido: o LABEL acessado pelo comando de controle de *workflow* não foi definido.

3.4 functions.pm

Módulo auxiliar que reúne as funções criadas em outros módulos da Máquina Virtual. Com ele é possível ter acesso às funções que podem ser executadas pela RVM.

Para cada comando implementado nos módulos listados, as funções recebem como argumento o objeto do pacote sobre o qual a ação deve ser realizada, o argumento (se não existir, é *undef*, e é descartado) e o número de elementos da pilha.

Todas as funções retornam a atualização do número de elementos da pilha, estritamente maior que 0, ou números negativos, indicando as exceções listadas na seção anterior.

3.5 f_arit.pm

O f_arit.pm contém as funções aritméticas que o programa deve executar, como a soma, subtração, multiplicação e divisão.

Ele desempilha o último e penúltimo valores a partir do topo da pilha de dados. Caso não existam dois valores, a função lança uma exceção.

Observação: As 4 operações funcionam de maneira análoga.

3.6 f_io.pm

Este módulo é responsável pela execução da função PRN (*PRiNt*), que imprime na STDOUT o valor do topo da pilha de dados.

3.7 f_jumps.pm

O f_jumps.pm contém as funções responsáveis pela execução dos comandos associados a jumps, como JMP (*JuMP*), JIT (*Jump If True*)e JIF (*Jump If False*).

No caso de *JMP*, este irá atribuir o argumento passado ao registrador de instrução (PC), realizando um salto incondicional a esta posição. Caso não existam instruções na posição indicada, lança uma exceção do tipo *Falha de Segmentação*.

Para os outros comandos de salto condicional (*JIT*/ *JIF*), também é possível passar como argumento um LABEL. Caso esta posição não exista, lança uma exceção específica.

3.8 f_memo.pm

O f_memo.pm contém as funções que mexem com endereçamento de memória: *STO* e *RCL*.

Este módulo é o responsável por realizar a transferência de dados entre o vetor de memória principal (DATA) e o vetor auxiliar (RAM). Cada um deles recebe como argumento o endereço (posição do vetor), de modo que:

- STO: retira o valor de DATA e coloca na posição $arg de RAM;
- RCL: realiza o processo inverso.

Ambas as funções devolvem uma exceção do tipo *Falha de Segmentação* caso não haja elementos em DATA (*STO*) ou na posição de RAM (*RCL*).

3.9 f_stk.pm

O submódulo f_stk.pm contém as funções de manipulação de pilha. *PUSH*, *POP* e *DUP*.

- PUSH: Recebe um argumento e o coloca no topo da pilha de dados DATA;
- POP: Retira o dado que estiver no topo da pilha de dados;
- DUP: Empilha uma cópia do topo da pilha.

3.10 f_tests.pm

O submódulo f_tests.pm armazena as funções lógicas. Cada uma dekas retira os dois elementos do topo da pilha e, em seguida, realiza uma comparação, usando como primeiro argumento o topo da pilha.

Considerando 'A' o topo da pilha e 'B' o elemento anterior, os comandos realizam as seguintes ações, em seus análogos para números:

- EQ: A == B
- GT: A > B
- GE: A >= B
- LT: A < B
- LE: A <= B
- NE: A != B

4 Packages

Foram criados pacotes para facilitar a divisão e entendimento do código.

4.1 Arena

O package "Arena"cuida das classes que envolvem os terrenos e a arena em si, bem como o classe que cuida da autorização de execução de determinadas funções que necessitam da consultoria da arena para serem executadas.

4.1.1 Action.java

Aqui é onde encontramos a implementação das funções utilizadas diretamente pelos robôs, tais como HIT, DRAG, DROP, MOVE, SEE e LOOK.

e tratamos a saída de modo que os dados que sejam emitidos na pilha ao usar a função, são devolvidos em um vetor de Stackables. Caso a função só lance uma String como saída, essa String será passada para tipo Text e preencherá a primeira casa do vetor de Stackables.

O restante das funções são auxiliares, que devolvem os respectivos resultados de cada função

4.1.2 Appearence.java

É uma estrutura chamada Enum, onde dizemos que as aparências podem ser somente as mostradas a seguir: DIRT, GRASS, ROCKY, ICE, WATER, SAND, JUNGLE e TUNDRA.

4.1.3 Map.java

A classe mapa vai cuidar da matriz de terrenos, e podemos assumir que cada mapa terá um clima diferente.

É nessa classe que também criamos os times a partir do parâmetro número do time passado, onde temos uma matriz representando jogadores de determinado time.

Também inserimos exércitos e removemos Scenários. Como nas seguintes funções:

public Scenario removeScenario(int i, int j) que eliminará os Scenários do terreno com as coordenadas passadas como parâmetro. Essa função serve para o caso de termos ataque a árvores e até mesmo a morte de robôs, já que estes também são do tipo Scenario.

public Robot insertArmy(String name, int player, int ID, int i, int j, String pathToProg) que irá inserir um robô de acordo com os parâmetros recebidos, como nome do robô, a qual jogador ele pertence, o seu ID e as suas coordenadas. Essa função irá criar esse robô e implementá-lo no como um Scenario nas coordenadas indicadas da arena.

4.1.4 Robot.java

Essa classe é a que define as características dos robôs em geral, como velocidade ro robô, vida, quantos danos ele causa, entre outras.

Dentre suas funções, temos o contrutor que recebe determinadas características e cria um robô, têm também a

public Item removeSlots(int position) que remove o robô de determinado terreno, a

public void identify() que imprime as características principais do robô,

public void run() executa o programa, acrescentando 1 no PC,

public void upload(Vector<Command> PROG) carrega um novo programa e

public int takeDamage(int damage) que, por ser um Scenario, remove das vidas do robô o dano sofrido.

E quando imprimimos um determinado robô, é devolvida uma String com o nome dele. Além de possuir também os getters e setters de determinadas características.

4.1.5 Terrain.java

Cada terrain é um empilhável e possui um tipo de rugosidade, uma aparência e possivelmente um Scenario.

A maioria de suas funções, além do construtor (que irá receber a aparência e a partir dela, determinar o tipo de rugosidade), tem-se basicamente getters, setters e removedores de características da classe. É

válido lembrar que utilizamos polimorfismo para o cnstrutor dessa classe, pois há vários tipos dessa função que recebem diferentes tipos e números de argumentos.

4.1.6 Type.java

Assim como em Appearence, type também é um Enum, só que dessa vez, representa os tipos de terreno da arena. Sendo eles, por enquanto: NORMAL e ROUGH.

4.1.7 Weather.java

Assim como Appearence e Type, Weather é do tipo Enum, representando desta vez. os vários tipos de clima da arena. Sendo eles:
CONTINENTAL, ARTICAL, DESERTIC e TROPICAL.

4.1.8 World.java

4.2 Exception

O package "Exception"cuida dos lançamentos de exceção que o programa gera para erros no código que será recebido do usuário.

Como por exemplo as exceções de falha de segmentação, operações inválidas, erros de tipo, variáveis não inicializadas, entre outros.

As classes desse package são chamadas pelas outras classes que utilizam essa forma de mandar exceção.

4.3 Operation

4.3.1 Operation.java

4.4 Parameters

4.4.1 Debugger.java

Essa classe irá auxiliar no encontro dos erros do cógigo, pois mostra passo a passo, como o programa do usuário está sendo lido e interpretado.

4.4.2 Game.java

É uma interface que detém as características do jogo que serão passadas para muitas outras classes. Tais como a velocidade da animação, o tamanho do mapa, o número máximo de robôs, entre outras.

Ela foi criada para ter um amior controle das características configuracionais do jogo e caso necessitem ser alteradas, todas estarão no mesmo lugar.

4.5 Parser

4.5.1 Parser.java

O parser recebe um vetor de trincas (comando, argumento, label), analisa as variáveis conforme o tipo correspondente no java e monta uma classe com o mesmo nome do programa em assembly.

Adicionalmente, o programa cria um arquivo "Parser.xml"para ser rodado pelo Ant e compilar todos os códigos no diretório do package.

4.6 Random

Aqui criamos os mapas de determinados cenários, gerados de forma probabilística, de modo que o mesmo cenário se altere de acordo com a semente passada. Cada cenário usa os tipos Scenario que são os elementos que compõe os cenários.

4.6.1 CalmField.java

Gera um cenário com árvores, rochas, pedras e cristais no terreno de forma probabilística. Não podemos esquecer que ela gera as bases necessárias para cada time.

4.6.2 Desert.java

Gera um cenário com rochas, pedras, cristais e bases de uma forma aleatoriezada e tenha as características de um deserto.

4.6.3 Jungle.java

Gera um cenário com árvores, rochas, pedras, cristais e bases. Também possui um rio gerado e distribuido probabilisticamente em meio a arena.

Esse rio possui partes de água mais rasas e outras mais profundas, são as chamadas deep water e water respectivamente.

4.6.4 RamdomMap.java

É responsável pela geração do mapa. Recebe parâmetros indicando qual será o tipo de clima e os associa com o respectivo tipo de mapa.

Em sua outra função, ela gera o mapa, colocando os determinados símbolos nos seus terrenos.

4.6.5 Theme.java

Essa é uma interface que padroniza todos os tipos de temas, como JUNGLE, DESERT e WINTER. Esse tipo de dado possui uma função específica que retorna uma matriz representando o mapa criado.

4.6.6 Winter.java

Gera um cenário com árvores, rochas, pedras, cristais e bases de forma aleatória. Além de gerar um rio congelado (água) no centro da arena.

4.7 Robot

Esse package cuida das funções atreladas aos robôs si, entre elas está a RVM e as funções que interpretam os comandos utilizados pelos usuários.

4.7.1 Arit.java

Essa classe controla as funções que interpretam os comandos aritméticos que o usuário passar na pilha de comandos, como por exemplo: Soma, subtração, divisão, Multiplicação e Resto. Tudo isso é manipulado na pilha de dados, recebemos os valores de lá, e mandamos a resposta para o topo da mesma.

Temos a seguinte função:

private static final void calculate(RVM rvm, Operation op) que cuida da generalização do trabalho realizado na pilhaEssa classe controla as funções que interpretam os comandos aritméticos que o usuário passar na pilha de comandos, como por exemplo: ADD (Soma), SUB (Subtração), DIV (Divisão), MUL (Multiplicação) e MOD (Resto). Tudo isso é manipulado na pilha de dados, recebemos os valores de lá, e mandamos a resposta para o topo da mesma. de dados pelas funções aritméticas.

4.7.2 Check.java

Essa função tem como objetivo checar o terreno em si e sua visinhança, bem como

static void ITEM(RVM rvm) que dá um pop na pilha caso o topo seja um terreno e empilha os itens contidos nesse terreno na pilha de dados.

static void SEEK(RVM rvm) que verifica se o topo é um Stackable e desempilha , depois verifica se o topo é um Around, e desempilha. Depois, percorre a visinhança do próprio robô (que dependerá do quão longe o robô enxerga) e tenta encontrar Items nessa visinhança.

Isso auxilia quando o robô for procurar cristais, por exemplo, fazendo com que ele não precise estar necessariamente na casa do cristal para notar a sua presença.

4.7.3 Command.java

Cria objetos do tipo comando, para serem manipulados da mesma forma na pilha de comandos. Tem como parâmetros o próprio comando, os argumentos do tipo Stackable e a label que caracterizam as linhas passadas pelo usuário no programa que o mesmo cria em baixo nível. Na classe, também temos as funções básicas getters e setters de determinadas características de cada comando.

4.7.4 Ctrl.java

É um comunicador que interpreta um comando do programa do usuário em forma de String (por exemplo: "JMP") e chama a respectiva função, passando os seus parâmetros para ela.

Dentre esses comandos que ela interpreta, estão também aqueles que podem ser executados pelos robôs, como MOVE, HIT, DRAG, DROP, SEE e LOOK.

4.7.5 Func.java

É responsável pelas funções CALL e RET que controlam o valor do PC quando comandos do tipo Jump são executados. Temos as seguintes funções:

static void CALL(RVM rvm, Stackable arg) Atualiza o PC para o valor que foi solicitado pelo por algum dos tipos de Jump, esse valor pode ser tanto uma label quanto um número que representa a linha do programa criado pelo usuário.

static void RET(RVM rvm) Essa função trabalha em conjunto com o CALL, já que ela recupera o valor que o PC tinha antes de ser alterado pelo próprio CALL.

4.7.6 IO.java

Essa classe controla as funções que interpretam os comandos IO (Input/Output) que o usuário passar na pilha de comandos, que no caso, só temos o print (static void PRN(RVM rvm)).

Nessa função, desempilhamos o topo da pilha e imprimimos seu valor.

4.7.7 Jumps.java

Assim como as classes Arit e IO, essa classe também executa os comandos passados pelo usuário, só que, desta vez, com funções Jump. Como o JIT (Jump If True), JIF (Jump If False), JMP (JuMP) e JCMP (Jump CoMPare), atuando de tal forma que o PC é atualizado para o endereço passado.

4.7.8 Mem.java

Da mesma forma que algumas classes anteriores, essa função cuida dos comandos passados pelo usuário. Sendo esses comandos, formas de controlar o armazenamento e recuperação de dados na memória.

Temos as seguintes funções:

static void STO(RVM rvm, Stackable position) que armazena o que está no topo da pilha de dados, na memória de acordo com o parâmetro recebido, que representa o endereço de memória em que se deseja ser salvo o dado. E após esse processo, retira o dado da pilha de dados.

static void RCL(RVM rvm, Stackable position) que recupera o que foi armazenado em um determinado endereço de memória, que é passado pelo usuário como parâmetro, e joga esse dado na pilha de dados.

4.7.9 Prog.java

Controla as funções que não tem uma função específica no programa, como o END (que não faz mais nada além de finalizar o programa) e NOPE (que não realiza nada).

Em: static void END(RVM rvm) atualizamos o PC para -1 (local onde não há nenhum outro comando) o que faz com que o programa seja finalizado.

4.7.10 RVM.java

Essa classe, mesmo que indiretamente, controla a pilha de dados e de comados do usuário, o PC,

4.7.11 Stk.java

Cuida das funções que mexem diretamente com a memória, como PUSH, POP e DUP. Que implementam um dado no topo da pilha, o retiram e duplicam o q está no topo, respectivamente.

4.7.12 Syst.java

Essa é a classe que controla os comandos destinados aos próprios robôs, como por exemplo HIT, DRAG, DROP, MOVE, SEE e LOOK.

Sendo que private static void action(RVM rvm, String type) guarda o tipo de comando passado da pilha de dados, cria uma Operação op e verifica se essa operação é válida, caso seja, coloca na pilha de dados as informações necessárias que serão lançadas como resposta.

4.7.13 Tests.java

Controla as funções que executam os comandos do usuário e que são do tipo comparativas. Entre elas, estão CMP, EQ, NE, LE, LT, GE, GT que dão pop no topo da pilha de dados, duas vezes, e compara os dois dados de acordo com a função especificada.

Depois desse processo, é empilhado um dado do tipo Num (que é um Stackable) na pilha de dados. De tal forma que, caso a comparação seja verdadeira, é empilhado um Num de conteúdo 1, caso contrário, um Num de conteúdo 0.

4.7.14 Var.java

Essa classe cuida dos comandos voltados a criação de variáveis no programa do usuário. Tais como alocação de memóiria, liberação de memória, receber e alterar o valor da variável. Temos as seguintes funções:

static void ALOC(RVM rvm, Stackable name) que recebe o nome da variável como parâmtro e verifica na RVM se não existe outra com outro nome. Caso negativo, é criada uma variável com valor null.

static void FREE(RVM rvm, Stackable name) que remove a variável com o nome name da RVM.

static void SET(RVM rvm, Stackable name) que pega o valor do topo da pilha e armazena na variável com o nome name, caso ela exista.

static void GET(RVM rvm, Stackable name) que pega o valor da variável e o coloca no topo da pilha de dados.

4.8 Scenario

Nesse package, temos as classes que representam os vários tipos de elementos contidos nocenários do jogo. Para o controle desses elementos. usamos a interface Scenario que além de padronizar o tipo de dado, força todas as classes a terem uma função que cuida dos danos adquiridos ao decorrer do jogo (por exemplo, quando uma pedra é atacada por um robô, ela perde vidas, e perdendo um determinado número de vidas, ela é destruida.

Dentre eles há a base, as rochas, árvores e água.

4.8.1 Base.java

A base tem um número escolhido ao acaso (42) de vidas, portanto, quando formos retornar o HP, sempre será 42. Já que a base, ao ser atacada, não recebe danos. E ao imprimirmos uma base, teremos como resultado "(ß) Base".

4.8.2 Rock.java

A pedra retira o dano recebido do seu HP(vida) e o devolve quando é chamada a função que retorna o HP. E ao imprimirmos uma pedra, teremos como resultado "(O) Rock".

4.8.3 Scenario.java

Interface que padroniza os diferentes tipos de elementos do cenários em um tipo Scenario, além de padronizar a função que cuida dos danos recebidos por cada objeto.

Tem-se as seguintes funções:

public int getHP() Retorna o número de vidas do presentes no robô.

public int takeDamage (int damage) Essa é a função que cuida dos danos recebidos pelo objeto, recebendo o dano como parâmetro para poder subtrair vidas do cenário.

4.8.4 Tree.java

Assim como a pedra, a árvore recebe danos e os subtrai de seu HP. E quando a imprimimos através das funções IO, temos "( ) Tree".

4.8.5 Water.java

A água, assim como a base, não recebe dano. Portanto ela tem um número específico de HP que se manterá intancto aos ataques sofridos. Ao imprimirmos uma árvore, temos "( ) Water".

4.9 Stackable

Nesse package, temos a interface Stackable que será o tipo responsável pela atribuição da característica empilhável dos empilháveis.

4.9.1 Addr.java

Os objetos do tipo Addr são endereços de determinados valores em um vetor. Portanto são representados por inteiros.

Nele encontramos as seguintes funções:

public Addr(int address) que é o construtor de objetos do tipo Addr.

public int getAddress() que retorna o valor do endereço como um inteiro.

public String toString() que além de retornar a String com o valor de da variável, manda o texto para as funções IO.

4.9.2 Around.java

Tem como característica uma matriz de Strings que representam as direções as quais os robôs devem seguir para chegar aos determinados pontos visinhos.

Nela, encontramos as seguintes funções:

public Around(Terrain[] seeing)

public String[] indexToPosition (int index) é responsável pelo retorno da String de coordenadas que o robô deve seguir para chegar a sua visinhança. Ela recebe como parâmetro, o número de casas de distância que o robô pode atacar/ver.

public void print() função para modo debuger!

public String toString() devolve a String "around", incluindo para saídas de IO.

4.9.3 Attack.java

Os objetos do tipo Attack são os tipos de ataque que o robô pode realizar. Por enquanto, temos o ataque MEELE que é a curta distancia(somente poderá atacar o que estiver nas casas visinhas a sua) ou RANGED que é a uma distância determinada pelas características do robô (caso o robo tenha um poder de atacar a 3 de distância, ele irá atingir o que está a até 3 casas de distância da sua).

Nela, temos as seguintes funções:

public Attack(String s) um construtor que monta o objeto a partir da String passada.

public String getAttack() retorna o tipo de ataque em forma de String.

public String toString() que além de retornar a String com o valor de da variável, manda o texto para as funções IO.

4.9.4 Direction.java

Os objetos do tipo Direction são do tipo que orientará os robôs na arena em relação a movimentação e ataque. Pelo fato de a arena ser formada por bases hexagonais, temos as direções E (leste), NE (nordeste), NW (noroeste), W (oeste), SW (sudoeste) e SE (sudeste). Sempre que um objeto do tipo Direction for chamado pelo usuário, este deve ser retratado com o símbolo "->" antes do seu conteúdo.

Por exemplo, "->WE".

A classe Direction possui armazenará o movimento como uma matriz de coordenadas, que terá determinados tipos de movimento quando o robô estiver em uma linha par e outro tipo de movimento quando o mesmo estiver numa linha ímpar. Essas coordenadas são dadas de modo que, quando a somamos com as coordenadas do robô, ele se movimentará para o local desehjado. Obs.: Quando a linha é par, usamos a 1ª linha da matriz, caso contrário, usamos a 2ª.

Temos as funções:

public Direction(String dir) que é o contrutor

public Direction(int move, int dir) que recebe a direção e a partir dela, preenche a matriz com as respectivas direções.

public String toString() que além de retornar a String com o valor de da variável, manda o texto para as funções IO. static void ALOC(RVM rvm, Stackable name) public int[] get(int row) Devolve um vetor com a direção a ser seguida, se a linha em que ele está for par, devolvemos a "( ) Tree". 1ª linha da matriz, caso contrário, a 2ª.

private void set(int even_x, int even_y, int odd_x, int odd_y) preenche a matriz com as direções que são recebidas como parâmetro.

4.9.5 Item

Item não é uma classe, é uma pasta com as classes que representam objetos do tipo item (implementam a interface de itens), o tipo item é um empilhável que pode ser pego e destruído pelo robô. Dentre elas, temos a

Crystal.java que é repsonsável pela criação de objetos cristais.

a Item.java que é uma interface responsável pela atribuição do tipo item às classes.

e a Stone.java que é responsável pela criação das pedras que são destruíveis e colecionáveis pelos robôs.

4.9.6 Num.java

É um empilhável do tipo número. Ela possui as seguintes funções:

public Num(double num) que é o construtor, que cria o objeto usando um parametro numérico que é recebido.

public double getNumber() que retorna o valor de Num em formato double.

public String toString() que além de retornar a String com o valor de da variável, manda o texto para as funções IO.

4.9.7 Stack.java

Essa classe administra as pilhas do programa, controlando informações sobre onde está o seu topo ou qual o seu tamanho.

4.9.8 Stackable.java

É a interface Stackable que se responsabiliza pelos objetos empilháveis. Essa interface não possui características exclusivas como funções e variáveis que seriam herdadas pelas classes desse tipo, é apenas para generalizar o tipo dos empilháveis.

4.9.9 Text.java

É criado o tipo de Stackable Text, onde poderemos empilhar um texto ou String. Possui as seguintes funções:

public String getText() que retorna o texto em si em formato String.

public String toString() que além de retornar a String com o valor de da variável, manda o texto para as funções IO.