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 com 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 da

4.1.1 Action.java

4.1.2 Appearence.java

4.1.3 Map.java

4.1.4 Robot.java

4.1.5 Terrain.java

4.1.6 Type.java

4.1.7 Weather.java

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

4.3.2 package-info.java

4.4 Parameters

4.4.1 Debugger.java

4.4.2 Game.java

4.4.3 Verbosity.java

4.5 Parser

4.5.1 Parser.java

4.6 Random

4.6.1 CalmField.java

4.6.2 Desert.java

4.6.3 Jungle.java

4.6.4 RamdomMap.java

4.6.5 Theme.java

4.6.6 Winter.java

4.7 Robot

4.7.1 Arit.java

4.7.2 Check.java

4.7.3 Command.java

4.7.4 Ctrl.java

4.7.5 Func.java

4.7.6 IO.java

4.7.7 Jumps.java

4.7.8 Mem.java

4.7.9 package-info.java

4.7.10 Prog.java

4.7.11 RVM.java

4.7.12 Stk.java

4.7.13 Syst.java

4.7.14 T ests.java

4.7.15 Var.java

4.8 Scenario

4.8.1 Base.java

4.8.2 Rock.java

4.8.3 Scenario.java

4.8.4 Tree.java

4.8.5 Water.java

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) é 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 1^a linha da matriz, caso contrário, usamos a 2^a.

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.

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 1^a linha da matriz, caso contrário, a 2^a.

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 dotipo 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() return this.text; que retorna o texto em si em formato String.

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