Relatório Exercício Programa II - Batalha de Robôs

Fellipe Souto Sampaio *
Gervásio Protásio dos Santos Neto †
Vinícius Jorge Vendramini ‡

29 de outubro de 2013

MAC 0242 Laboratório de Programação II
Prof. Marco Dimas Gubitoso

Instituto de Matemática e Estatística - IME USP
Rua do Matão 1010
05311-970 Cidade Universitária, São Paulo - SP

*Número USP: 7990422 e-mail: fellipe.sampaio@usp.com
†Número USP: 7990996 e-mail: gervasio.neto@usp.br
‡Número USP: 7991103 e-mail: vinicius.vendramini@usp.br

1 Introdução

Esse relatório se destina a explicar a implementação do Segundo Exercicío-Programa da disciplina de Laborátorio de Programação II.

Nesta etapa do projeto foram desenvolvidos os alicerces sobre os quais fundamentaremos etapas posteriores.

Foi desenvolvida um máquina virtual capaz de interpretar código em "linguagem de máquina"e todos os elementos necessários para que fossemos capazes de extrair informação relevante do código e executá-lo.

Desenvolveu-se também uma interface (e uma classe derivada) que representa o prótotipo dos robôs, essencialmente interpretadores de código fonte movidos pela supracitada máquina virtual.

Como cenário de jogo desenvolveu-se uma Arena, que funciona como um controlador dos robôs.

Por fim, como especificado no enunciado foram inclusos chamadas ao sistema (SysCalls) que permitiriam uma interação mais profunda e complicada entre a maquina virtual do robô e a Arena.

2 Arena

2.1 Apresentação

A classe Battlefield é a classe principal, responsável por gerenciar os componentes do programa, como a interface gráfica e a execução das instruções dos robôs.

2.2 Atributos

Um dos atributos principais da classe é o map, que é essencialmente um mapa do campo. Ele é uma matriz de chars, com cada char representanto um hexágono do terreno; a partir desses chars é que desenhamos o mapa. Os chars são determinados a partir de "defines"no começo do código, e a criação da matrz é feita de forma relativamente gráfica no começo do main, para permitir uma fácil edição de qualquer aspecto do mapa (e até a futura criação de outros mapas).

O map é usado principalmente na criação da MatrixHexa. Essa matrix é essencialmente uma matriz de Terrains, que são chars também. Ela tem basicamente a mesma informação da map, mas como é um objeto ela permite futuras sofisticações de terrenos ou do mapa que nós queiramos fazer.

A arena é uma matriz do mesmo tamanho do mapa, mas usada para controlar as posições das entidades do jogo, principalmente para facilitar o output. Ela é uma matriz e Entity's, que no momento podem ser robôs ou cristais, dependendo do char type dela. Dessa forma, basta varrer essa matriz para saber onde devemos imprimir robôs ou cristais.

Finalmente, temos a matriz screen. Essa matriz contém literalmente o que vamos mostrar na tela, então é bem maior do que as outras; cada char dela será impresso no output. Esse é um sistema bem simples, pois é apenas um modo de visualizar a situação do jogo enquanto não criamos o output gráfico de verdade.

2.3 Métodos

A classe tem poucos métodos para a criação do mapa, que são basicamente o main, dois inicializadores e um método para imprimir a tela.

O main é bem simples. Ele começa inicializando o map, de forma que fica fácil fazer qualquer mudança no terreno, desde o seu tamanho até de cada hexágono que o compõe. Depois disso, criamos a MatrixHexa a partir desse mapa e chamamos os dois inicializadores, para então imprimirmos tudo na tela.

initArena é o método que inicializa a Arena, a matriz de Entity's. Ele basicamente cria a matriz, popula ela com espaços em branco e depois insere robôs e cristais aleatoriamente pelo campo.

initScreen já é mais complexo, pois envolve desenhar os hexágonos. A orientação dos hexágonos foi escolhida de modo a tornar melhor a representação deles em caracteres. Inicialmente, método determina o tamanho da tela a ser impressa e cria a matriz, e então popula ela com espaços em branco. Em seguida, coloca os limites do mapa, ou seja, os hexágonos que são apenas parcialmente impressos no canto da tela e servem só para fins estéticos. Em seguida, imprimimos os contornos dos hexágonos, para poder facilmente distinguir entre uma célula e outra. Retiramos os pequenos traços que ficariam soltos nos cantos superior esquerdo e inferior direito e então preenchemos os terrenos, com base

na MatrxHexa. Com a base do mapa já criada, resta apenas desenhar os robôs e os cristais. Para isso simplesmente percorremos a Arena, e a cada robô ou cristal encontrado desenhamos ele na célula certa. Os robôs são da forma de "stick-figures"(desnhos de palito), enquanto os cristais são apenas asteriscos '*'.

Com tudo pronto, basta apenas imprimir a tela, o que se torna muito fácil com a matrix screen. O método printArena apenas percorre screen, imprimindo cada caracter encontrado e quebrando a linha conforme necessário.

Esse modo de fazer o output torna fácil, se necessário, a animação do jogo, porque basta atualizar o mapa e a arena como desejado, chamar novamente initScreen e printArena e temos uma nova tela impressa.

3 Stack - a pilha

A pilha da máquina virtual, que guarda todos os dados relevantes para as operações dos robôs, teve sua implementação dividida em dois códigos fontes:

1. **StackableIterface**: define métodos comuns a todos os tipos empilháveis. Ambos conseguem indetificar seu valor e imprimir-se na tela. Esses tipos são:

   (a) *Stackable_String*: strings que podem ser colocadas na pilha.

   (b) *Stackable_Number*: números sobre o qual a pilha é capaz de fazer operações.

2. **StackElemente**: É a pilha propriamente dita junta com objetos que possibilatam a simulação de um ciclo FDX (Fetch-Decode-Execute) na Máquina Virtual (detalhada na seção 5. Contém um objeto do tipo *MemoryArray* (essencialmente um vetor dinâmico moldado às necessidades especiais do projeto) que funciona como memória e é acessado pelos métodos RCL e STO.

   A pilha em si, ou seja, a estrutura de dados onde armazenamos objetos empilháveis é feita por meio de um ArrayList (um vetor dinâmico nativo do Java).

   Além disso, a classe StackElement também contém um objeto do tipo *Operation* responsável por operações aritméticas e lógicas. Sua presença permite que todas as operações relevantes a serem realizadas com objetos empilháveis possam ser feitas na própria classe StackElement.

   Por meio de métodos próprios da classe e por métodos da classe *Operation* é possível realizar operações de pilha e logico-aritiméticas, possibilitando a execução de um código-fonte.

4 Máquina Virtual

A classe VirtualMachine é a maquina virtual *de facto*. É responsável pela interpretação e execução de código-fonte.

Contém um objeto da classe *Parser*, responsável por receber uma string contendo o nome de um arquivo de código fonte, e fazer a análise léxica e sintática do código, construíndo um vetor programa, com as instruções a serem executadas e seus respectivos argumentos. Para isso, usa-se fortemente a biblioteca RegularExpressions do Java, para facilitar identificação de padrões do texto (do código-fonte) e extrair os dados relevantes.

O vetor contendo o programa (que é montado durante a chamada do método *parseToMe* da classe *Parser*) também é um atributo da máquina virtual, sendo responsável por conter o fluxo do programa.

Outro atributo (também formado durante a chamada ao supracitado método da classe *Parser*) é uma Hashtable, responsável por guardar pulos do código (os argumentos de comandos como JMP). A key do hash é o *label* que identifica o trecho de código ao qual deseja-se ir; e associada à key está a posição do início deste trecho do vetor programa.

Um componente integral da VM é a pilha (um objeto da já descrita *StackElement*. Por meio deste objeto, é possível realizar a execução do código ditado pelo vetor programa.

Por fim, como elementos auxiliares, temos um inteiro que guarda que posição do código é executada do presente e usamos Enumerators para definir tanto as ações a serem executadas quanto o estado atual da VM.

5 SysCalls

SysCalls é a forma com que a maquina virtual faz solicitações de uma determinada ação ao sistema, podendo alterar um elemento do cenário (TAKE), um elemento exterior (FIRE), alterar a si mesmo ou perguntar sobre sua condição (LOOK, ASK, WLK). Durante uma rodada de execução das instruções de um robô o sistema operacional está aguardado que a entidade faça alguma solicitação. Para evitar que a execução trave em apenas um sujeito o sistema permite que um número máximo de instruções seja executadas, e caso nenhuma chamada ao sistema seja feita antes desse máximo ser atingido uma interrupção é provocada e a maquina é colocada em modo *WAITING* até uma nova rodada. O modo *WAITING* é um dos três estados possíveis que a maquina virtual/robô podem ficar, sendo cada um deles :

- RUNNING - O a maquina virtual está executando seu conjunto de instruções.
- CALLING - A maquina virtual encontrou uma chamada ao sistema ou excedeu o limite de instruções processadas.

- WAITING - A maquina está aguardando uma nova rodada de processamento.

Como o ciclo de execução dos robôs é algo intermitente até o fim da partida não existe fim no processamento do vetor de instruções, e para isso o program counter(PC) deve ser resetado sempre que atingir o fim do arquivo[1].

6 Robôs

Os robôs são basicamente implementações robustas da máquina virtual. Uma interface genêrica, a **Robot.java** que define métodos que permitem lidar com o robô e acessar qualidades básicas dele, tais como vida, posição e time. São definidos também métodos setters, que funcionam de forma a iniciar as condições do robô.

Atualmente, a única classe a implementar esta interface é a classe *BattleRobot.java*, à qual, no presente momento do projeto, todos os robôs pertencem. Além de implementar concretamente os métodos da interface, essa classe possui ainda uma Máquina Virtual, iniciada com o código fonte que se deseja rodar inicialmente, durante a construção do Robô.

Por meio do método *runVm* executa-se um ciclo da máquina virtual, o que essencialmente permite ao robô executar comandos, funcionando com um interpretador da linguaguem assembly definida. Além das características citadas o robô ainda possui outros atributos como modelo, número de série (este é gerado aleatoriamente durante a fase de criação do robo, é por esse número que o sistema reconhece o robô), vida, coordenadas no mapa, seu estado atual, seu nome e seu time. Uma importante característica é o time a qual um robô pertence. Com intuito de viabilizar o funcionamento do programa nesta fase decidiu-se que cada time deveria ter um mesmo conjunto de instruções (sourcecode) comum, facilitando assim o entendimento de como cada robô se comporta nas execuções de suas instruções e das chamadas ao sistema[2].

7 Design, Implementação e Funcionamento

Nesta fase optamos por implementar as ferramentas básicas para a modelagem da Arena e da entidade robô. Algumas partes serão remanejadas para a fase seguinte, seja por dificuldade de implementação ou também por decisão de integração com a interface gráfica. Dentre das etapas que deixaremos para próxima fase destacamos as seguintes:

> Alterar a forma como é inserido as instruções das VM, pretende-se criar um diretório com diversos códigos e em cada código estará explícito a qual robô aquele código fonte deve ser atribuido.
>
> Melhorar o iterador dos robôs para que as entidades que chegaram ao fim de seu código (código END) possam continuar sendo processadas.

[1] Em processo de implementação para as próximas fases
[2] Maiores explicações na seção "Design, Implementação e Funcionamento"

Dar um peso a cada chamada do sistema, transformando a fila em uma fila de prioridades aleatorizada.

Ampliar a interface robô e criar mais robôs derivados dela.

Refinar o funcionamento dos estados da VM.

Implementar a execução das chamadas ao sistema, não só sua identificação.

Nosso programa funciona dentro das limitações que citamos ao longo do relatório, todavia, é possível testar e verificar a criação dos robôs, a execução das intruções, as requisições de chamadas ao sistema e parte de seu processamento e por fim a criação e impressão da arena.

7.1 Complilação e execução

Para compilar utilize o makefile que está dentro da pasta com o comando *make* via terminal. A classe principal é Battlefield.java e para executar o programa é necessário 2 arquivos sourcecode, um para cada exército de robôs. As instruções válidas são as descritas ao longo do enunciado, e as chamadas dos sistema possiveis são:

WLK

BOMB

FIRE

TAKE

LOOK

ASK

todas estas intruções possuem como argumento uma string válida (sem o caracter # no meio). Por fim uma execução possível seria a seguinte : *java Battlefield sourceCodeA sourceCodeB* . Com esta chamada será exibido informações sobre todos os robôs, o processamento de seu código, as chamadas ao sistema processadas pela fila e por fim a impressão do mapa. O programa acompanhará uma pasta de exemplos testados e reposta obtida.