Batalha de robôs

Marco Dimas Gubitoso

4 de setembro de 2013

Sumário

1 Introdução

O jogo se passa em uma arena ou mundo habitado por exércitos formados por robôs virtuais. Os robôs são autônomos e obedecem a um programa interno, redigido pelos jogadores e que pode ser substituído a qualquer momento. O objetivo é colecionar 5 cristais especiais e levá-los até a base dos exércitos inimigos. O exército que tiver os 5 cristais colocados em sua base estará automaticamente fora do jogo. O último exército a permanecer na arena é o vencedor.

A descrição que segue é intencionalmente vaga em diversos pontos, para que possamos discutir em classe e em uma *wiki* especialmente criada para isso no Paca.

As próximas seções descrevem os principais elementos do jogo, que serão detalhadas em momento oportuno. O servidor (sistema de gerenciamento) e o cliente serão componentes do sistema, se houver tempo, implementaremos o jogo em rede.

1.1 Arena

A arena nada mais é do que uma região onde a batalha ocorre. Internamente é uma matriz *hexagonal* de terreno, onde estão descritas as posições das bases, dos cristais e tipos de solo e acidentes geológicos.

Cada elemento da matriz pode ser um dos seguintes tipos (outros poderão ser acrescentados):

- Terreno plano — o robô pode entrar e sair com custo mínimo.
- Terreno rugoso — o custo de saída é 3 vezes maior.
- Repositório de cristais — contém um certo variável de cristais, mas são inicialmente invisíveis. Um robô só poderá ter ciência da sua posição após explorar o sítio.
- Base — a base de um exército, o ponto que deve ser defendido.

1.2 Robô

Como foi dito, um robô é uma unidade autônoma, isto é, não precisa de comandos do usuário para agir. Seus modos de ação são programados a priori. Isto faz com que ele seja um interpretador de uma linguagem, implementando uma máquina virtual.

Esta máquina será capaz de enviar solicitações ao sistema de gerenciamento do jogo (veja a seção 1.3), informando seu desejo em andar, atacar, explorar, etc. O resultado de cada ação dependerá do andamento do jogo todo. Cada chamada retornará a nova posição do robô e seu estado.

1.3 Sistema de gerenciamento

A parte central do jogo é o sistema de gerenciamento. É ele que mantém o estado da arena, trata das requisições dos robôs e dos jogadores. Em essência, é um servidor associado a um mecanismo de atualização de estados.

Na sua versão final, o servidor deverá executar as seguintes tarefas:

1. Inicializar a arena, seja criando uma arena nova a cada jogo, ou lendo um cenário pronto do disco.
2. Aguardar conexões dos jogadores e, para cada um deles:
   (a) Definir uma base na arena
   (b) Carregar os exércitos e distribuí-los na arena.
   (c) Enviar os dados completos do jogo, assim que definidos.
3. Iniciar um laço que permanecerá em execução até que o jogo termine. Cada iteração tratará de um passo de andamento do jogo (*timestep*). Este passo compreende diversas ações:
   (a) Verificar e tratar chamadas especiais dos jogadores (desistência e alteração de programa do robô).
   (b) Tratar requisições dos robôs.
   (c) Reposicionar os elementos do jogo.
   (d) Enviar dados de atualização de cenário para os jogadores (clientes)

1.4 Cliente

O programa cliente é o responsável pela interface com o usuário e a conexão com o sistema gerenciador. Do ponto de vista de tarefas, ele é relativamente simples, suas atribuições são as seguintes:

1. Permitir que o usuário configure seu exército, programando os robôs e distribuindo atributos de energia, força, velocidade, etc.
2. Fazer a conexão e registro com o servidor.
3. Apresentar a arena graficamente para o usuário, com todas as informações relevantes.

4. A cada passo, receber do servidor as atualizações do jogo e alterar a imagem mostrada ao jogador de acordo.

5. Permitir que o jogador faça as solicitações ao servidor.

2 Primeira fase (em Perl)

A primeira fase, feita em *perl*, não será usada na versão final do projeto. No entanto, ela será bastante útil para aprimorar as ideias e automatizar os testes, até que o compilador esteja completo. A critério de cada um, ela poderá ser adaptada para auxiliar o desenvolvimento de outras formas.

Esta fase é composta de duas partes: a implementação de uma máquina virtual, que posteriormente será substituída por outra em *java*; e um montador, que lê um arquivo fonte e gera código executável na máquina virtual.

2.1 A máquina virtual

A máquina virtual irá reger o comportamento dos robôs. É necessário definir os tipos de variáveis que esta máquina pode manipular e quais as instruções fundamentais e avançadas disponíveis para o programador. Felizmente a implementação é simples e a inclusão futura de novos tipos e instruções é fácil, como veremos.

A máquina se baseia em uma pilha de dados, como em uma calculadora pós-fixa ou RPN. Além disso, ela possui uma pilha de execução e um vetor de memória. Algumas variáveis especiais poderão ser manipuladas diretamente pelo programa, veja abaixo.

As instruções são colocadas sequencialmente em um vetor e uma variável inteira marca o ponteiro de execução, isto é, o índice da instrução sendo executada.

Recapitulando, cada máquina virtual possui, pelo menos, as seguintes variáveis:

Vetor com o programa Um vetor com a sequência de instruções que devem ser executadas.

Ponteiro de instruções Um escalar inteiro com a posição da próxima instrução a ser executada. É um índice do vetor de programa.

Pilha de dados Uma pilha com os dados usados na execução do programa. Em *perl* é simplesmente mais um vetor.

Pilha de execução Uma pilha com endereços de retorno, para chamadas de funções.

Memória Simplesmente um vetor com valores.

2.1.1 Instrução

Uma instrução nada mais é do que um par (*opcode, valor*)[1], onde o *opcode* é uma constante indicando o tipo de operação e *valor* é um operando que pode não ser necessário, dependendo da instrução específica.

2.1.2 Tipos de dados

Os tipos que devem ser aceitos na máquina virtual são os seguintes:

- Número
- Ação
- Cristais
- Terreno
- Vizinhança
- Endereço de variáveis

Outros podem ser incluídos, de acordo com o interesse de cada grupo e com as discussões na *wiki*.

Cada tipo corresponde a uma classe, que deverá ter um construtores específicos, com todas as possibilidades de argumentos cabíveis.[2]

2.1.3 Conjunto de instruções

O conjunto de instruções é o mais delicado em termos de escolha, pois define o que a máquina poderá executar ou não. Além de operações básicas, colocaremos algumas instruções complexas, para facilitar a programação.

[1] usei o anglicismo *opcode* porque acredito que deixa a descrição mais clara.

[2] Números podem ser representados diretamente, sem a necessidade de uma classe em separado. Isso mudará na versão em java.

Instruções básicas Este é um subconjunto minimal e não é específico para o jogo, mas é necessário para permitir a interpretação de uma linguagem mais completa:

- Manipulação da pilha
- Operações aritméticas
- Desvios
- Chamada e retorno de funções
- Atribuição e consulta a variáveis

Em todos os casos, os operandos, se houver, devem ser verificados quanto à compatibilidade da operação.

Manipulação da pilha As operações normais de pilha, acrescidas de instruções auxiliares úteis:

- Empilha — coloca um *Empilhável* na pilha
- Desempilha — retira e retorna o topo da pilha
- Dup — duplica o topo
- Descarta — retira o topo
- Inverte — troca a ordem dos dois elementos no topo
- Consulta — retorna uma cópia do topo da pilha, sem retirá-lo

Operações aritméticas As operações usuais de soma, subtração, etc. Podem ser incluídas as funções mais interessantes, ou mesmo operações novas que se mostrem úteis de alguma forma.

Operações lógicas De modo similar às aritméticas, as operações lógicas atuam sobre os valores no topo da pilha, empilhando o resultado (*verdadeiro* ou *falso*).

Desvios Aqui se encontram os desvios incondicionais e os condicionados ao valor no topo da pilha. O operando é o deslocamento com relação à posição atual do ponteiro de execução.

Chamada e retorno de funções Para simplificar a chamada de funções, usaremos uma pilha adicional, a *pilha de execução*, que conterá apenas os endereços de retorno. Desta forma não precisaremos nos preocupar com a implementação do quadro (*frame*). Os argumentos são empilhados normalmente na pilha de dados e cabe à função retirá-los, se necessário.

A operação de retorno simplesmente desvia para o endereço no topo da pilha de execução. Se a função precisar devolver um valor, ela simplesmente o coloca na pilha de dados antes de retornar.

Instruções específicas São instruções que não se enquadram nos casos anteriores, como término de programa, por exemplo. Para testes, é interessante incluir uma instrução que imprime o topo da pilha. Novas instruções podem ser incluídas posteriormente. Vamos discutir na *wiki*.

As instruções que devem ser implementadas neste momento estão relacionadas na descrição do montador

2.1.4 Execução

A execução simplesmente percorre o vetor de programa, usando o ponteiro de instruções e executa a ação correspondente ao *opcode* encontrado. Estipularemos que a execução sempre se inicia na posição 0 do vetor.

2.2 Montador

A segunda parte desta fase é o montador, responsável por traduzir um texto com um código fonte (*assembly*) e gerar o vetor de programa descrito acima.

O formato da entrada é bastante simples, consistindo de uma série de linhas com a seguinte estrutura:

$$[label\textbf{:}]\ [opcode\ [argumento]]$$

Os []s indicam que os campos são opcionais, além disso, valem as seguintes regras:

- Comentários se iniciam com # e seguem até o final da linha.

- Linhas vazias são ignoradas.
- Cada linha com *opcode* corresponde a uma posição no código do programa.
- Um *label* define uma constante com a posição corrente do programa.

2.2.1 Código de instruções

As instruções que devem ser reconhecidas e consequentemente implementadas na máquina virtual, nesta fase, são as descritas a seguir. Exceto onde explicitado, as instruções atuam sobre a pilha de dados.

- `PUSH` empilha seu argumento.
- `POP` descarta o topo da pilha.
- `DUP` duplica o topo da pilha, isto é, empilha uma cópia do topo.
- `ADD` desempilha dois argumentos e empilha sua soma.
- `SUB` desempilha dois argumentos e empilha sua diferença (subtrai o topo do segundo elemento).
- `MUL` desempilha dois argumentos e empilha seu produto.
- `DIV` desempilha dois argumentos e empilha a razão entre o segundo elemento e o topo.
- `JMP` atribui o seu argumento ao ponteiro de instruções.
- `JIT` *jump if true* atribui seu argumento ao ponteiro de instruções se o topo da pilha for verdadeiro. Em qualquer caso, descarta o topo.
- `JIF` *jump if false* atribui seu argumento ao ponteiro de instruções se o topo da pilha for falso. Em qualquer caso, descarta o topo.
- `EQ` desempilha dois argumentos e empilha o resultado da comparação de igualdade.
- `GT` similar, para comparação de valor maior entre o
- `GE` similar, para maior ou igual.

- `LT` similar, para menor.
- `LE` similar, para menor ou igual.
- `NE` similar, para diferença (não igualdade).
- `STO` remove o elemento do topo e armazena no vetor de memória, o índice é dado pelo argumento da instrução.
- `RCL` empilha elemento do vetor de memória que se encontra na posição dada argumento da instrução.
- `END` término da execução.
- `PRN` desempilha e imprime o topo da pilha.

O programa final desta fase deverá ler um arquivo fonte, gerar o vetor de instruções e executá-lo.

Exemplos de programas

Conta simples

```
INIC:   PUSH   10
        PUSH   4
        ADD
        PUSH   3
        MUL
        PRN
        END
```

Fibonacci

```
# inicializa
        PUSH   1
        STO    0     # x
        STO    1     # y
        PUSH   10
        STO    2     # i
LOOP:   RCL    0
        RCL    1
        DUP
        STO    0     # x' = y
        ADD          # x+y
        DUP
        STO    1     #y = x+y
        PRN
        RCL    2
        PUSH   1
        SUB          #i-1
        DUP
        STO    2     # i = i-1
        PUSH   0
        EQ           # i == 0?
        JIF    LOOP
        END
```