Batalha de robôs

Marco Dimas Gubitoso

20 de novembro de 2017

Sumário

1 Introdução

O jogo se passa em uma arena ou mundo habitado por exércitos formados de 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 número 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

A primeira fase é 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 em *C* e um montador em *Python*, 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 *C* é simplesmente mais um vetor.

Pilha de execução Uma segunda 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.

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

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 *struct*, que deverá ter um construtores específicos, com todas as possibilidades de argumentos cabíveis.

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.

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***:**] [*opcode* [*argumento*]]

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

- 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.
- `CALL` Empilha o endereço da próxima instrução na *pilha de execução* e desvia para seu argumento.
- `RET` Empilha seu argumento na *pilha de dados*, desempilha o endereço da *pilha de execução* e desvia para ele.
- `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
```

2.3 O que entregar

Veja nos arquivos anexos os programas quase completos. O seu grupo deverá incluir os seguintes elementos:

- Uso de variáveis locais para funções. Estas variáveis ficarão na ***pilha de execução***, de forma parecida com o que ocorre no *assembler*. Para isso, a máquina virtual necessita de um outro registrador de base, que fará o papel do `rbp`.
- Instruções para acesso a variáveis locais. São duas novas instruções: `STL` e `RCE`.
  - `STL` remove o elemento do topo e o armazena na pilha de execução, o índice é dado pelo argumento da instrução somado ao valor do registrador de base.
  - `RCE` empilha elemento do vetor de memória que se encontra na posição da pilha de execução dada argumento da instrução somado ao registrador de base.
- A instrução `RET` deve acertar o valor do registrador de base antes de desempilhar o endereço de retorno.
- O montador completo, isto é, gerando um arquivo equivalente ao "`motor.c`", com uma máquina virtual e com o código gerado. Esta saída deve ser compilável e o programa resultante deve funcionar corretamente.

3 Segunda Fase — A Arena e a Máquina Virtual

O desenvolvimento desta fase dependerá de conceitos que serão apresentados em classe, nas próximas aulas. Mesmo assim, já é possível projetar o que deve ser feito e tratar da implementação e reconhecimento das instruções adicionais.

3.1 A Máquina virtual (robô)

Uma versão da máquina já foi construída na primeira fase, o que deve ser feito é sua adaptação para chamadas ao sistema e implementação do estado.

Um dos problemas que precisam ser enfrentados é a identificação do tipo de variável que está no topo pilha, para o tratamento correto na execução. Isto pode ser feito de algumas formas diferentes, o mais simples é definir uma *struct* nova com dois campos: indicador do tipo; e valor.

Precisaremos também de instruções novas para fazer chamadas ao sistema e consultas especiais.

3.1.1 Atributos adicionais

Para o jogo, a máquina precisa contar com novos atributos:

- Posição: um par de valores x e y indicando a posição do robô na Arena.
- Cristais: quantidade de cristais carregados pelo robô.

Opcionalmente podem ser colocados outros itens, como *vida*, *energia* ou *combustível*.

Não havará instruções para manipular estes valores, já que isto implicaria em mudança direta no estado do jogo. Todas as operações sobre estas variáveis só poderam ser feitas por meio de chamadas ao sistema (veja a seguir).

3.1.2 Tipos de dados

Começaremos a trabalhar com tipos estruturados de dados, pois em alguns cass a informação pode ser mais complexa. Por exemplo, em uma célula da Arena, precisamos descrever vários elementos:

- Terreno — tipo do terreno (estrada, montanha, rio, etc). Dependendo do valor, o custo para entrar ou sair pode ser maior ou menor.
- Cristais — número de cristais estão presentes nesta célula
- Ocupação — já existe um robô presente nesta célula?
- Base — é uma das bases dos exércitos? Qual?

Para ter acesso a estes campos, precisamos de uma nova instrução: `ATR`, que lê um dos atributos do objeto colocado no topo da pilha. O atributo a ser lido é dado pelo argumento:

```
ATR 3
```

retira o objeto do topo da pilha e empilha seu quarto atributo.

3.1.3 Operandos das instruções

Em todos os casos, os operandos, se houver, devem ser verificados quanto à compatibilidade da operação. Isso pode ser feito pela máquina virtual verificando o tipo de cada valor da pilha.

3.1.4 Chamada de sistema

Precisaremos incluir uma instruções especiais para fazer chamadas ao sistema (Arena), solicitando operações que alterem o estado do jogo (pegar ou depositar um cristal, movimentar o robô, ataques, etc).

Cada instrução especial deverá colocar os argumentos na pilha de dados e chamar a Arena (veja em 3.2), obtendo uma resposta. No retorno, o resultado se encontrará no topo da pilha.

3.2 Arena

Contém a representação do mundo. Essencialmente é uma matriz hexagonal[2] cujos elementos são identificadores de terreno como descrito na seção 1.1, mas deve também conter a posição dos robôs e o estado geral do jogo.

Entre as informações que a Arena deve manter, encontram-se:

- Lista dos exércitos ativos, com a posição de cada robô.
- Tempo transcorrido

Note que os dois primeiros atributos são objetos compostos, portanto devem ser implementados em uma *struct* específica.

Os métodos importantes, além do construtor, incluem:

- `Atualiza()` — movimenta os exércitos e atualiza o estado do sistema. É o método responsável para avançar um *timestep*. nesta implementação, significa fazer com que cada robô se adiante um certo de número de ciclos da máquina virtual. Isto é facilmente implementável, pois o método `run` da máquina virtual já serve para executar o *timestep*
- `InsereExercito()` — inclui um novo exército no jogo.
- `RemoveExercito()` — retira um exército derrotado do jogo.

[2] discutida em classe

- `Sistema(int op)` — Este método será chamado por uma instrução especial da máquina virtual. O argumento possui uma identificação do tipo de chamada e eventuais operandos estão colocados na pilha de dados, como explicado acima. Para exemplos, veja 3.2.1.

A Arena pode ainda percorrer sua matriz e atualizar os diversos elementos de terreno, permitindo um mundo dinâmico. Discutiremos isso em classe.

3.2.1 Chamadas ao sistema

As chamadas ao sistema permitem a alteração do estado do mesmo. A solicitação específica está codificada na *Operação* que, como já dito, possui um código, operando e uma referência ao robô que realizou a chamada. Os resultados deverão ser tratados (por exemplo, empilhar) pela instrução que realizou a chamada.

Note que um robô não pode modificar seu estado sozinho, a menos de variáveis internas que não tem impacto direto sobre o ambiente. Se ele quiser se mover, deverá solicitar ao sistema que faça isso.

Veja alguns exemplos:

- `[Mover, direção]` — o código mover solicita que o robô seja movido na direção indicada. A direção é uma das seis possibilidades de deslocamento em uma matriz hexagonal.
- `[Recolhe,direção]` — recolhe um cristal que está posição vizinha indicada pela direção.
- `[Deposita,direção]` — deposita um cristal na posição vizinha indicada pela direção.
- `[TipoAtaque,direção]` — realiza um ataque do tipo indicado na direção. O alcance do ataque depende de seu tipo.

3.3 Detalhes finais

É possível que surjam conflitos entre requisições ao sistema: por exemplo, dois robôs tentam ir para a mesma posição. Uma saída é colecionar todos os movimentos e decidir caso a caso. No entanto, vou propor uma solução mais simples em aula.

Nesta fase, o mais importante é ter o arcabouço montado. Nem todas as chamadas do sistema precisam ser implementadas e mesmo que uma ou outra instrução fique faltando, não haverá problema contanto que seu programa esteja preparado para colocá-las rapidamente nas próximas fases.

3.4 O que entregar

O sistema básico para a Arena, com o escalonador e a chamadas de sistema. No momento, as chamadas apenas mostrarão resultados e não irão resolver conflitos, isto será feito na próxima fase.

O módulo da Arena deve conter:

- Construção da matriz que representa a Arena e sua inicialização (mais detalhes em aula).
- Regristro de máquinas virtuais: a Arena deverá conter um vetor de ponteiros para máquinas virtuais de tamanho suficientemente grande, digamos 100. A função de registro deve receber o endereço de uma máquina virtual e copiá-lo na primeira posição livre deste vetor.
- O escalonador. É simplesmente uma função que percorre o vetor de máquinas e manda cada uma executar um número pequeno (50) de instruções virtuais, retornando à primeira após percorrer o vetor todo. Na primeira versão, o escalonador deverá receber um parãmetro dizendo quantas rodadas serão executadas. Na próxima fase ele deverá verificar uma condição de parada.
- Modificações na máquina virtual para incluir os atributos adicionais explicados em 3.1.1.
- Testes para todas as novas funcionalidades.

Veja que apesar de parecer muita coisa, o trabalho é relativamente simples. Veja mais dicas em aula.

4 Terceira Fase — Sistema de Jogo e Apresentação Gráfica

A segunda fase montou o funcionamento básico da máquina virtual e do sistema de gerenciamento. Agora completaremos o sistema de jogo, a menos

da linguagem de alto nível, que ocorrerá na última etapa.

O trabalho necessário para esta fase é um pouco menor, já que a parte mais complicada já está pronta. A maior novidade é a interface gráfica, que será apoiada por exemplos de implementações, disponibilizadas em aula e no Paca.

4.1 Sistema de jogo

O sistema de jogo nada mais é do que a Arena com os robôs e demais elementos já distribuídos e com todas as chamadas de sistema implementadas.

Seguem os pontos que devem estar implementados para que o jogo comece a funcionar como tal.

4.1.1 Robôs

Os robôs devem ser inicializados de forma completa. Existem variáveis de estado importantes, como saúde, energia, tipo e número de armas. As variáveis que serão implementadas depende da complexidade desejada do jogo.

O robô básico deve ter pelo menos a saúde (percentual de avaria), um nível de "ocupação" (veja abaixo) e um objeto para guardar a vizinhança apresentada pelo sistema.

Robôs podem ser sofisticados para incluir outras características, como velocidade em cada tipo de terreno, energia, sensores, etc. Fica a critério de cada grupo o que implementar, mas a implementação não deve impedir extensões a priori.

O mais importante é que o robô *não pode mudar seu próprio estado pela máquina virtual*. Qualquer mudança de estado é feita pela Arena.

4.1.2 Arena

Na inicialização, a Arena deve realizar as seguintes tarefas:

- Inicializar a matriz que representa o mundo, indicando o tipo de terreno em cada célula (veja abaixo)
- Determinar as células bases de cada exército e marcá-las
- Distribuir os cristais aleatoriamente

- Distribuir os exércitos de robôs, tomando o cuidado de não sobrepô-los. Esta distribuição pode ser aleatória ou não, é uma opção de cada grupo. Minha sugestão é colocar os robôs em uma área próxima à sua base.

Após a inicialização da Arena, o mundo construído deve ser apresentado graficamente (seção4.2) de modo que as células mostrem apropriadamente os itens gráficos correspondentes (o terreno específico, a presença ou não de um robô, etc).

Após esta inicialização, as máquinas virtuais devem começar a executar os programas recebidos, uma instrução por vez, e prosseguirão iterativamente. As máquinas serão percorridas por um iterador aleatório.

Ao escalonar a máquina virtual de um determinado robô, a Arena deve antes verificar se este robô está "ocupado". A ocupação é determinada por um contador no robô. Se o contador for 0, a instrução pode ser executada normalmente, caso contrário, o contador é decrementado e a instrução não é executada. Isto permite que chamadas ao sistema durem mais do que um ciclo, permitindo penalizar o robô quando caminhar por terrenos escorregadios ou quando estiver recebendo um novo programa na última fase.

4.1.3 Chamadas ao sistema

Como já dissemos, toda alteração de estado deve ser feita exclusivamente pela Arena[3]. Ações como ataque, movimentação, pegar ou largar um cristal, devem estar implementadas por chamadas ao chamadas ao sistema (seção 3.2.1).

Algumas, talvez todas, chamadas já foram implementadas na segunda fase. Nesta fase, é o momento de completá-las e verificar seu funcionamento correto. Em resumo, cada chamada deve:

1. Verificar o tipo de operação solicitada.

2. Verificar a viabilidade da operação: o caminho pode estar obstruído na direção do movimento, a saúde do robô não permite que ele carregue mais cristais, ou não existem cristais na posição, etc.

3. Se a operação puder ser executada, atualiza o estado do sistema de acordo.

[3]Não custa lembrar que Arena e "sistema" se referem à mesma entidade, que controla a execução do programa todo

4. Atualiza, se for o caso, o contador de ocupação do robô.

5. Retorna.

4.2 Apresentação gráfica

A apresentação gráfica será feita por um programa em *Python*, veja o material anexado a esta fase. A comunicação com python será feito por um *pipe*, usando a função `popen`, descrita nos exemplos.

Essencialmente o visualizador recebe comandos pela sua entrada padrão, dizendo o que deve ser desenhado. No material em anexo há um exemplo de como fazer isto, além do programa de visualização.

4.2.1 Visualizador em Python

Veja o programa `apres` no anexo.

Para abrir uma janela e desenhar, usaremos a biblioteca `pygame`. É bem provável que você precise instalá-la. No Linux, isto pode ser feito pelo gerenciador de pacotes da sua distribuição ou, caso não esteja disponível, com o programa `pip3`[4]:

```
pip3 install pygame
```

Além de `pygame`, também usaremos `fileinput`, que já faz parte da distribuição normal do python e que serve para ler da entrada padrão.

Os comandos recebidos pela entrada padrão formam um *protocolo*, que está parcialmente implementado no exemplo, mas que precisará ser estendido para acomodar todas as exigências. No exemplo, as linhas aceitas são estas:

- `fim` — termina a exibição e fecha a janela. O mesmo acontece de a entrada for fechada.

- `rob` *arquivo* — registra um robô com o desenho indicado pelo arquivo, que deve ser uma imagem. Os robôs serão identificados posteriormente pelo seu número de ordem de registro (começando por 0).

- *id x_orig y_orig x_dest y_dest* — Usada para movimentar um robô. São 5 inteiros representando, respectivamente, a identificação do robô,

[4]ou `pip`, se você usar a versão 2.7, mas os programa foi testado na 3.5

posição de origem e posição de destino. Veja que o programa não faz consistências, apenas redesenha a célula de origem (sem recolocar o robô) e desenha o robô sobre a célula de destino.

4.2.2 Controle pelo programa principal

Veja o programa `controle.c` no anexo. Este programa tem mais coisas do que o necessário, serve para testar a visualização. Na prática, quem mandará os comandos para a visualização serão as chamadas de sistema que alteram o estado.

A parte essencial do programa são apenas 5 linhas, espalhadas pelo código:

```
FILE *display;

   display = popen("./apres", "w");

   fprintf(display, <COMANDO>);

   fflush(display);

   pclose(display);
```

Veja que *display* funciona como um arquivo normal, para escrita. A diferença é que usei `popen` e `pclose` ao invés de `fopen` e `fclose`. Esta mudança faz com que o programa `apres` no diretório corrente seja iniciado e sua entrada padrão seja conectada à `display`. Ou seja, tudo o que for escrito em `display` será alimentado na entrada padrão de `apres`. O `fflush` força que cada escrita seja enviada imediatamente.

4.3 O que entregar

Primeiro é preciso completar sua arena para que todas as chamadas sejam implementadas (caso não estejam ainda). As chamadas que mudam o estado, como posição de robô e números de cristais, por exemplo, devem incluir o envio de um comando para o programa de visualização (`apres`).

Além disso, deve ser feita a inicialização da arena, com indicação de terrenos, posição de exércitos, bases e cristais. Isto implica em enriquecer o protocolo de comunicação com o visualizador. Estes são os novos comandos, mas você pode colocar mais, se assim desejar:

- `cristais` n i j — coloca n cristais na célula (i, j).
- `base` *imagem* i j — coloca uma nova base, indicada pelo desenho *imagem* na célula (i, j). *imagem* é um arquivo, como no caso dos robôs.

Outras possibilidades são indicar o tipo de terreno que deve ser desenhado na célula e colocar mais detalhes nos robôs. Por exemplo, ao colocar um robô, passar também seu nível de energia e o número de cristais que está carregando. Naturalmente o programa `apres` deve ser alterado de acordo.

O programa `apres` está praticamente pronto, mas talvez seja necessário mudar a função `convert` para adequar à sua representação da matriz hexagonal. Coloquem dúvidas no fórum.

5 Quarta Fase — Programando o robô em alto nível

Para descrever a estratégia do robô, a programação na linguagem da máquina virtual é muito inconveniente, pois é preciso se concentrar no conteúdo da pilha e nas instruções de baixo nível. Usaremos portanto uma linguagem de alto nível, que torna a programação mais confortável e com meios mais poderosos.

Precisaremos de um compilador, que traduzirá esta linguagem para a máquina virtual. Para isso, usaremos o *bison*, que é uma ferramenta que lê uma especificação de gramática e gera o compilador correspondente.

O primeiro passo é definir a linguagem.

5.1 A linguagem

A linguagem deverá ter os comandos usuais de controle (condicionais e laços), definição e chamadas de função, variáveis e comandos especiais para o jogo (criação de ação, leitura dos atributos do robô e da arena, etc).

Uma versão semi-pronta da linguagem está disponível no PACA, contendo inclusive a definição e chamada de funções, bem como o tratamento de variáveis locais. Falta colocar o tratamento de erros, variáveis especiais e comandos específicos. O "`else`" não está implementado, mas é fácil colocar. **Atenção:** a máquina virtual implementada segue uma definição ligeiramente diferente, faça as adaptações necessárias.

5.1.1 Gramática

A gramática deverá ser a seguinte, usando a notação do *Bison*:[5]

```
Programa: Comando
        | Programa Comando
        ;

Comando: Expr EOL
        | Cond
        | Loop
        | Func
        | RET EOL
        | RET OPEN  Expr CLOSE EOL
        | EOL
        ;

Expr: NUM
    | ID
    | ID ASGN Expr
    | Chamada
    | Expr ADD Expr
    | Expr SUB Expr
    | Expr MUL Expr
    | Expr DIV Expr
    | '-' Expr %prec NEG
    | OPEN Expr CLOSE
    | Expr LT Expr
    | Expr GT Expr
    | Expr LE Expr
    | Expr GE Expr
    | Expr EQ Expr
    | Expr NE Expr
    ;

Cond: IF OPEN  Expr CLOSE  Bloco ;
```

[5]discutam possíveis extensões na *wiki*.

```
Loop: WHILE OPEN  Expr CLOSE Bloco

Bloco: ABRE Comandos FECHA ;

Comandos: Comando
    | Comandos Comando
    ;

Func: FUNC ID OPEN Args CLOSE  Bloco ;

Args:
    | ID
    | Args SEP ID
    ;

Chamada: ID OPEN  ListParms  CLOSE ;


ListParms:
    | Expr
    | Expr SEP ListParms
    ;
```

Esta gramática deve ser estendida para incluir os comandos especiais, como criar ação, varrer a arena, etc. A varredura da arena pode ser uma instrução da máquina, ou um laço gerado automaticamente pelo compilador.

Note que é possível incluir construções bem interessantes, com comandos criativos e primitivas especiais para o jogo. Experimente, uma vez que se tenha o compilador básico funcionando, não é difícil colocar mais coisas.

5.2 Execução

O exemplo mostra o compilador combinado com o programa principal e com um sistema de execução. Os laços não foram implementados no exemplo, veja dicas em aula. A versão do jogo deve ter estes elementos separados. O compilador receberá o vetor de instruções e o preencherá com o código compilado. A máquina virtual, embutida no robô, tratará de executar este vetor sempre que requisistada.

Deve ser feito o tratamento de erros, tanto da execução como da compilação.

A inicialização dos robôs, feita na terceira fase, deverá agora receber o arquivo com o código fonte, o resto continua igual.