Projeto de Laboratório de Programação II - Fase 1

Karina Suemi, Vinícius Silva, Renato Cordeiro

1 Introdução

Nesta fase inicial, o projeto consiste em criar um leitor e interpretador de comandos em linguegem de máquina. O principal objetivo destes é servir como núcleo da futura classe responsável pelos robôs.

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 - LE - LT - MUL - NE - POP - PRN - SUB
Numérico	PUSH - RCL - STO - JMP - JIT - JIF
String	JIF - JIT

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 d a 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