Relatório - Parte I, II e III da Aventura*

Emanuel Lima (9009493) e João Seckler (4603521)

5 de Dezembro de 2018

Para executar o jogo, faça:

```
$ make
$ ./jogo.out
```

Para executar rotinas de teste (partes I e II do projeto), faça:

```
\$ make test
```

Em seguida, para testar as bibliotecas de elementos, lista e tabela, ou para testar o funcionamento básico do jogo e de seus comandos, faça, respectivamente:

```
$ ./lib-test.out
$ ./jogo-test.out
```

O programa de teste deve imprimir uma mensagem na saída padrão se houver algum erro.

Para limpar os diretórios:

```
\$ make clean
```

1 Parte I

Optamos por definir dois tipos para elemento: "elemento"e "Elemento". O segundo é um ponteiro para o primeiro. A desvantagem é que precisamos criar duas funções "elemento_cria"e "elemento_destroi", mas em compensação nos parece que o restante do código, nos módulos de lista e tabelas, fica mais limpo e legível. Fazemos isso inspirados na implementação de um módulo de pilhas feita em MAC0121 - Algoritmos e Estruturas de Dados.

Com essa diferença, algumas funções têm sua declaração alterada:

```
Lista insere(Lista l, Elemento *val)
```

torna-se Lista

*Trabalho para a discplina MAC0216 - Técnicas de programação I, ministrada pelo professor Marco Dimas Gubitoso no segundo semestre de 2018.

```
insere(Lista l, Elemento val);

Elemento *busca(Lista l, char *n)
```

torna-se

```
Elemento busca(Lista l, char *n); e

Elemento *retira(Lista l, Elemento *val)
```

torna-se

```
Elemento retira(Lista l, Elemento val)
```

Além disso, decidimos não seguir a sugestão do enunciado para a implementação da lista ligada à risca. Ele sugeria que o tipo lista fosse definido como:

```
typedef struct {
Elo * cabec ;
} Lista ;
```

No entanto, optamos por definir o tipo Lista simplesmente como um ponteiro para o tipo Elo. Assim, a cabeça de toda lista é criada pela função cria_lista, e é ela mesma um "Elo"cujo "val"não aponta para nenhum lugar relevante. Se é verdade que essa implementação gasta o espaço desse "val", que não é usado, também é verdade que evitamos novas definições, que pareciam tornar o código menos claro. O benefício apresentado pelo enunciado, qual seja, "o endereço da lista não muda com inserções e deleções", continua valendo, pela definição de "Lista". Essa estratégia é inspirada nas página sobre lista encadeada do Prof. Paulo Feofiloff (https://www.ime.usp.br/ pf/algoritmos/aulas/lista.html).

Na implementação da função

```
tabela_insere(Tabela T, char *n, Elemento val)
```

optamos por devolver um código de erro se o valor val associado à chave n já estiver armazenado no mesmo índice de T. Esse evento pode ter dois significados: (i) que o usuário tentou associar uma mesma chave a um mesmo valor mais de uma vez ou (ii) que o usuário associou duas chaves a um mesmo valor, e essas chaves colidem na função hash.

Implementamos a função hash da tabela de símbolos inspirados no seguinte material: http://www.cse.yorku.ca/ oz/hash.html

2 Parte II

2.1 Listas e tabelas

Uma alteração significativa que fizemos da primeira parte para a segunda foi nas bibliotecas de lista e de tabela. Agora, não são mais lista de *Elementos*,

mas são listas genéricas. Ou seja, o valor delas é agora um ponteiro genérico (**void*). Isso permite que sejam usadas nos mais vários tipo de situação.

Além disso, implementamos um sistemas de chaves já na biblioteca de listas (em que um nó da lista é composto de um valor, de tipo ponteiro genérico, uma chave, de tipo *string*, e um ponteiro para o próximo nó). Isso permite, primeiro, que a biblioteca de tabelas não precise de tanto código, pois não precisa implementar o sistema de chaves. Depois, permite que se realize busca a partir de chaves em situações em que a tabela seria uma estrutura demasiadamente grande, o que se mostrou útil na implementação do restante do jogo.

É boa prática não misturar endereços na memória que armazenam dados com endereços que armazenam instruções, ou seja, não misturar ponteiros com ponteiros de funções (na verdade, os compiladores nem permitem tal intercâmbio). Por isso, não seria possível que usássemos a biblioteca de listas, como estava feita, para fazer listas de funções. Contudo, precisávamos de listas que fizessem isso, pois seriam usada em todo o jogo, continuamente. A solução que encontramos, portanto, foi adicionar à biblioteca de lista novamente funções iguais às que já estavam lá, mas que serviam para ponteiros de funções. Ou seja, agora podemos criar, com essa mesma biblioteca, listas de ponteiros genéricos e listas de ponteiros de função genéricas (*void (*)(void)*). Esse segundo tipo de lista é usado no campo *acoes* dos elementos.

2.2 Comandos

A implementação dos comandos, ou verbos, se deu da seguinte maneira. Padronizamos o tipo das funções como:

```
Elemento funcao(Elemento, Elemento, Elemento);
```

Escolhemos esse padrão, mais complexo do que o sugerido nas instruções, porque optamos por não usar variáveis globais. Algumas funções exigem, portanto, três argumentos. Tome um exemplo: o verbo *pegar*. A função que o implementa terá que retirar o elemento da sala onde estava e colocá-lo no inventário do jogador. Para isso, terá que acessar o campo *conteudo* do elemento sala e do elemento jogador.

Num nível mais alto, existem os comandos genéricos. Eles são, por exemplo: *ir_para*, *examinar*, *comer*, *beber*, *inventario*, etc. Todas essas funções, de "alto nível", antes de executarem qualquer coisa verificam se o elemento que lhes foi passada como argumento tem, no seu campo *acoes*, um comando equivalente àquele. Por exemplo: seu eu chamo

```
comer(papel, NULL, NULL);
```

a função comer verifica se há, no campo *acoes* do elemento *papel*, uma função cuja **chave** é "comer". Havendo, ela a executa e não faz mais nada. Não havendo, executa algo padrão, que vale para todos os objetos (nesse caso, seria algo como imprimir uma mensagem que diz que você não consegue comer papel). A implementação disso é muito simples. Há, na biblioteca de comandos, uma

função chamada *papel_comer*. Na inicialização do objeto *papel*, essa função é inserida em seu campo *acoes* com a chave "comer".

Essa abordagem permite que nenhum comando que o usuário dê fique sem resposta (desde que haja **alguma** implementação para ele). Por mais absurdo que seja o que ele esteja pedindo, haverá sempre, no mínimo, uma mensagem padrão de erro.

Ela permite também que não precisemos implementar as saídas das salas como elementos. Suponha, por exemplo, que a porta da sala 1, onde está o jogador, para a sala 2 esteja trancada. Se o usuário quiser ir para a sala 2, ele dará o comando *ir_para*. Esse comando vai, antes de tudo, procurar na sala 2, em sua lista de verbos, se há lá uma versão de *ir_para*, e vai encontrar. Esse comando vai imprimir uma mensagem dizendo que a porta está trancada, e não vai deixar o jogador passar.

2.3 Rotina de testes

Para testar os elementos e os comandos, implementamos uma estrutura um tanto primitiva (afinal, ela vai ser substituída, na próxima fase, por código que envolve análise léxica e sintática). Ela consiste numa inicialização "à mão" de 4 vetores (à mão porque descrevemos os elementos desses vetores um a um). Um é um vetor de funções, os outro três são vetores de elementos. A combinação de um elemento de cada vetor é o que é necessário para executar um comando (ver 2.2). Assim, com os quatro vetores é possível executar uma série de comandos, o que é feito no laço principal do programa. Note que usamos mais quatro vetores de strings auxiliares, para poder imprimir na tela, a cada iteração, o que o programa está fazendo.

3 Parte III

Mais uma vez, vamos explicar as diferenças do nosso código para o que foi sugerido.

A principal diferença talvez seja um menor número de *tokens* na análise léxica. Enquanto a sugestão definia um token para cada tipo de elemento ou verbo (os tokens "VERBO", "OBJ" e "LUGAR"), optamos por colapsar esses três em um só. Um verbo fica definido como a primeira palavra do comando. Um lugar é um tipo de objeto, de qualquer jeito. Logo, definimos só um token, "OBJ". Mais tratamentos de erro acontecem nas funções chamadas. Mantivemos, como na sugestão, os *tokens* "SAIR", "INVENTÁRIO", "EOL" e aqueles de direção (norte, sul, lesto, oeste). Esses últimos funcionam, no nosso código, como uma espécie de "wrapper" para um comando de movimento (no nosso código, um comando de movimento é tratado como um comando qualquer, vide seção 2). Além disso, adicionamos um token "AJUDA", que imprime uma mensagem de ajuda.

Além disso, optamos por carregar menos de código aqueles arquivos de análise de linguagem. Um exemplo: quando recebemos inventário, simplesmente

chamamos uma função da biblioteca de *comandos*. Na sugestão que recebemos, esse código estava no próprio arquivo *.y*.

Por fim, optamos por não usar mais variáveis globais além das que *Flex* e *Bison* já usam. Precisávamos, no entanto, que a interpretação gramatical alterasse variáveis do programa principal. Para isso, portanto, usamos uma opção do bison chamada "%parse-param", que define parâmetros para a função *yyparse* (que é chamada no programa principal).