# Aventura!

Gubi

20 de novembro de 2018

## Sumário

# 1 Introdução

O objetivo do projeto é montar um sistema de execução de jogos do tipo *adventure*, como apresentado em classe. Neste tipo de jogo o personagem principal (aventureiro) executa ordens dadas pelo jogador e descreve o que aconteceu, sempre em forma de texto.

A estória acontece em um mundo virtual composto de lugares interligados por passagens, transferências mágicas, teletransporte, etc. Neste mundo existem vários objetos (virtuais, lógico) espalhados e que podem interagir com o aventureiro de diversas formas (veja a seção 2.1.2).

Existem alguns jogos disponíveis em diversos lugares:

- No emacs, execute `M-x dunnet`
- No linux, rode o programa `adventure`, que nas distribuições *Ubuntu* e *Debian* fica no pacote `bsdgames.deb`. Este jogo é uma adaptação do primeiro escrito neste estilo, o "Colossal Cave", em FORTRAN!
- Outra possibilidade com o linux é o `battlestar`, também em `bsdgames.deb`
- Procure jogos da *Infocom*, que marcou época com seus jogos de texto interativo. No linux existem vários pacotes para interpretar estes jogos. Experimente o `frotz` e o `gargoyle-free`.
- Experimente o engraçadíssimo *The Hitchhikers Guide to the Galaxy*. Passarei mais detalhes na sala de aula.

O projeto será desenvolvido em C e terá as seguintes componentes fundamentais:

- Biblioteca com os elementos básicos para a construção do jogo.
- Mecanismo interno do funcionamento do jogo.
- Interface elementar com o usuário.
- Interpretador da linguagem (quase) natural e finalização.

Além disso, o grupo deverá montar uma estória, seguindo rigidamente a especificação adotada no curso. Estórias montadas por um grupo devem poder ser usadas por outro grupo sem a necessidade de uma adaptação.

# 2 Estrutura interna

Antes de prosseguir, vou adiantar um pouco da estrutura interna a ser usada na descrição do mundo e seus objetos. Mais adiante, após termos visto melhor a noção de objetos e herança, retornarei a este ponto e o aprofundarei.

O mundo é composto por **elemento**s, que por sua vez podem ser lugares ou objetos. Os elementos serão implementados por *struct*s em C, usando modularização. Usarei **objeto** para um objeto dentro do mundo virtual. Para ajudar a fixar, **lugar** indicará uma região do mundo virtual.

Além dos **elemento**s, a descrição do jogo ainda contém **saída**s e **verbo**s. As **saída**s são conexões entre **lugar**es e os **verbo**s descrevem as ações possíveis.

A **objeto** e **lugar** são derivações de **elemento**, já que sua estrutura é parecida, mas existem algumas diferenças que estão discutidas nas próximas seções. Todo **elemento** possui algumas propriedades que o descrevem, ou a seu estado, e pode ser alvo de um conjunto de ações. Os **elemento**s devem conter um nome interno, um ou mais apelidos, uma descrição longa e uma descrição curta.

Os apelidos indicarão como o usuário irá se referenciar ao elemento. Por exemplo um *struct* que descreve um gato pode ter nome "gt" e apelidos "gato", "felino" e "bichano". funcionar como apelido também.

Nesta fase trataremos os **lugar**es e os *struct*s de forma independente, embora em java eles sejam relacionados. Assim, no que segue, haverá alguma redundância.

## 2.1 Elementos

Tanto **objeto**s como **lugar**es possuem as seguintes propriedades. O termo entre parênteses é o que será utilizado na especificação para indicar a propriedade.

- Nome — identificação interna do **elemento**, deve ser um nome único (representa a variável associada).
- Artigos (artigos) — uma lista de quatro palavras, indicando os artigos: definido direto, definido indireto, indefinido direto, indefinido indireto. Por exemplo: "o do um dum".
- Descrição longa (longa) — texto que descreve detalhadamente o **elemento**, apresentada quando o usuário pede para examinar o **objeto** ou **lugar**.
- Descrição curta (curta)— descrição abreviada, sem detalhes. É a descrição normalmente apresentada.
- Lista de objetos (contem) — lista dos nomes dos **objeto**s presentes no **lugar** ou contidos no **objeto**. Opcional.
- Atributos — são valores genéricos que permitem especificar melhor o estado, de acordo com as necessidades do jogo. Opcionais.

- Ação (acao) — **verbo** que tem significado especial neste **lugar** ou para este **objeto**. É opcional e pode haver mais de uma.
- Animação (animacao) — **verbo** especial que é chamado a cada iteração, permitindo animações. É opcional.

### 2.1.1 Lugares

Um **lugar** possui a seguinte propriedade adicional:

- Saída (saida) — conexão para outro **lugar**. As direções possíveis são pelos apelidos de cada saída (ver mais adiante).

### 2.1.2 Objetos

Além das propriedades dos **elemento**s, os **objeto**s possuem ainda:

- Adjetivos (adjetivos) — lista adicional com adjetivos que permitem especificar melhor o **objeto**.
- Invisível (invisivel) — indicação especial para **objeto**s que estão escondidos no início do jogo.

As açoes genéricas para **objeto**s são as seguintes:

- `examinar` — descreve o objeto.
- `pegar` — passa o objeto para o aventureiro, se possível (podem haver restrições quanto a número, tamanho ou peso que o aventureiro pode carregar).
- `largar` — o **objeto** precisa estar com o aventureiro. O **objeto** é colocado no lugar onde o aventureiro se encontra. Com argumentos adicionais, pode-se colocar um **objeto** dentro de outro, se possível.

Ações específicas para alguns objetos podem ser `destruir`, `esfregar`, `ligar`, etc. Procure ter uma implementação padrão para verbos mais gerais. Na especificação da estória, o usuário poderá definir novos verbos.

## 2.2 Saídas

As **saída**s ligam um **lugar** a outro. Uma **saída** é um caso especial de **objeto**, que possui duas propriedades especiais a mais:

- Destino (destino) — contém a referência para o **lugar** onde ela leva.
- Fechada (fechada) — indica que a **saída** está fechada.

## 2.3 Verbo

Um **verbo** é essencialmente uma função que atua sobre os **elemento**s. Veremos sua descrição nas próximas fases.

## 2.4 O aventureiro

O aventureiro é um caso muito especial de **objeto** animado. Ele pode conter outros **objeto**s, se os estiver carregando ou vestindo, e recebe atualizações diretamente do usuário.

Além disso, podem haver outras atualizações automáticas, como cansaço, fome e recuperação, se a especificação do jogo indicar.

# 3 Biblioteca básica

Para construirmos tudo isso, precisamos de algumas estruturas fundamentais: uma tabela de símbolos que liga nomes a **elemento**s e listas de diversos tipos.

### 3.0.1 Tabela de símbolos

A tabela de símbolos deve ser implementada por uma tabela de *hash* (detalhes em aula), onde a chave é uma *string* e o valor é um ponteiro genérico.

Além da estrutura de dados pura, devem ser implementadas as seguintes funções:

- `TabSim cria(int tam)` — cria uma tabela com `tam` entradas.
- `void destroi(TabSim t)` — destroi a tabela `t`.
- `int insere(TabSim t, char *n, Elemento *val)` — insere o nome `n` na tabela `t` e o associa com o valor `val`. Deve retornar um código de erro, indicando se a inserção foi bem sucedida ou não.
- `Elemento *busca(TabSim t, char *n)` — retorna o valor associado a `n` na tabela `t`. Deve retornar `NULL` caso o valor não seja encontrado.
- `int retira(TabSim t, char *n)` — remove o valor `n` da tabela, liberando a memória.

### 3.0.2 Lista de valores

As listas devem ser implementadas como listas ligadas (detalhes em aula). Da mesma forma que a tabela de símbolos, a implementação deve conter um conjunto de funções:

- `Lista cria()` — cria uma lista vazia.
- `void destroi(Lista l)` — destroi a lista `l`.

- `Lista insere(Lista l, Elemento *val)` — insere o valor `val` na lista `l` retornando o endereço do elemento inserido, ou `NULL` em caso de erro. Veja em aula a razão de fazermos desta forma.
- `Elemento *busca(Lista l, char *n)` — retorna o valor associado ao nome `n` na lista `l`. Deve retornar `NULL` caso o elemento não seja encontrado.
- `Elemento *retira(Lista l, Elemento *val)` — remove o elemento `*val` da tabela, sem removê-lo da memória.

## 3.1 Programa da primeira fase

O programa da primeira fase deve conter os módulos com as estruturas acima e um modulo adicional para testes. Todas as funções devem ser testadas em várias situações, para garantir que a biblioteca não contém erros.

Junto com o programa, deve também ser entregue um relatório simples com instruções de como usar o programa, detalhes da implementação e lista de eventuais problemas encontrados.

## 3.2 Sugestões

Para esta primeira fase, defina uma *struct* para elementos, com informações mínimas:

```
typedef struct {
  char n[80];
} Elemento;
```

Nas próximas fases, colocaremos o resto das informações.

Para a lista ligada, é muito mais prático ter uma estrutura separada com um ponteiro para a cabeça da lista. Desta forma, o endereço da lista não muda com inserções e deleções:

```
#include "elemento.h"

typedef struct elo {
  struct elo *next;
  Elemento *val;
} Elo;

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

Para não tirar toda a diversão, deixo a tabela de hash inteiramente por conta de vocês. Há documentação *online* no próprio IME.

# 4 Segunda fase: Arcabouço Básico do Jogo

Nesta fase, montaremos a descrição de todos os elementos do jogo. Já poderemos testar a dinâmica do jogo, embora ainda não seja possível jogar.

Antes de discutir as estruturas, vejamos alguns pontos mais específicos da linguagem C.

## 4.1 Recursos especiais de C

C possui alguns recursos especiais, que estão presentes em várias outras, embora muitas vezes disfarçados. Você pode considerá-los como truques e o conhecimento de linguagem de máquina deve ajudar bastante em sua compreensão.

### 4.1.1 Definição de tipos

Para definir um tipo novo, ou dar um novo nome a um tipo já existente, basta colocar `typedef` antes da declaração de alguma variável. Com isto, o nome da "variável" passa a ser um novo tipo e pode ser usado como tal. Esta sintaxe permite simplificar em muito a utilização de tipos. Por exemplo, na definição de `struct`s, como já vimos.

Destaco aqui pois faremos bom uso do `typedef`.

### 4.1.2 *Unions*

Além de `struct`s, C também permite a criação de `union`s. A `union` difere da `struct` no fato de que todos os seus campos ocupam a *mesma* posição de memória. Enquanto o tamanho de uma `struct` é a soma dos tamanhos dos seus campos[1], o tamanho de uma `union` é o tamanho do seu maior campo.

Isso pode parecer estranho, pois significa que podemos escrever o valor de um tipo e ler como a representação de outro.

```
#include <stdio.h>
typedef union {
  long int i;
  float f;
} Duplo;
⋮
int main()
{
  Duplo d;
  d.i = 54798058876894908;
  printf("%g\n", d.f);
  return 0;
```

[1] Eventualmente o compilador pode colocar alguns *bytes* a mais para alinhamento

```
}
```

Colocando um valor em `d.i`, alteramos o valor de `d.f`. O valor de `d.f` tem a mesma representação binária de `d.i`, mas o valor é completamente diferente, pois `float` e `long int` são representados de formas distintas.

Para que serve uma bizarrice dessas? Uma aplicação é poder ler facilmente a sequência de *bytes* que representa um valor (como?). A principal é permitir um tipo que admita mais de um formato, veja a seguir.

Reflita sobre esta afirmação: se pensarmos que um tipo representa o conjunto de valores que uma variável pode assumir, a `struct` é o produto cartesiano dos tipos de seus campos. A `union` é (*surprise, surprise*) a união dos tipos de seus campos.

Suponha que queiramos um campo que pode ser um inteiro ou uma string, mas não os dois ao mesmo tempo. Claro que neste caso, seria conveniente ter outro campo indicando qual representação estamos usando.

```
typedef enum {
    False, True
} Boolean;

typedef union {
  char * descr;
  int idade;
} Idade;

typedef struct {
  char * nome;
  Boolean def;
  Idade id;
} Pessoa;

   ⋮

  Pessoa p1,p2;
  p1.nome = "Fulano";
  p1.def  = 0;
  p1.id.descr = "Indeterminada";

  p2.nome = "Ciclano";
  p2.def = True;
  p2.id.idade = 42;
```

### 4.1.3 Ponteiros especiais

Em C, os ponteiros carregam em sua definição o tipo de variável para o qual apontam. `Tipo *p` permite a aritmética de ponteiros especial `p+2` significa `p+2*sizeof(Tipo)`.

Para definir um ponteiro "genérico", o declaramos com o tipo `void`: `void *gp;`. para usá-lo, precisaremos fazer um *cast*, mudando seu tipo para o que efetivamente precisamos, por exemplo

```
void *pp;

 ⋮

double f = ((double *)pp)[2];
```

Pense como isso pode ser usado para construir listas ou tabelas de hash genéricas. Outro uso muito frequente é quando usamos `malloc`, que retorna um `void *`.

**Ponteiros para funções** O nome de uma variável do tipo *array* pode ser usado como seu endereço. É na verdade o *label* usado no *assembly*. O mesmo vale para os nomes de funções. Se quiser saber qual o endereço de memória ocupado por uma função, por exemplo `main`, basta fazer `printf("%lx\n", (unsigned long)main);`.

Ponteiros para funções são muito práticos, podemos passá-los como argumento para outras funções (procure a documentação do `qsort`).

Outra aplicação, que usaremos no projeto, é permitir que elementos do jogo possuam funções próprias, sem precisarmos definir um monte de `structs`. Basta ter um campo que aponte para a função desejada (ou para uma lista de funções).

Veja um exemplo de um ponteiro para função que recebe um inteiro e retorna um Elemento:

```
/* Tipo ponteiro para função */
typedef Elemento (*FPTR)(int);

FPTR f;
```

outro exemplo:

```
#include <math.h>

typedef double (*OSCILADOR)(double);
int main(int ac)
{
    OSCILADOR osc;
    double x;
    if (ac > 2) osc = sin;
    else osc = cos;
    x = osc(0.213412);
     ⋮
```

## 4.2 Definição dos Elementos

Os Elementos do jogo podem ser Objetos ou Lugares. Para isso usaremos uma `union`. A `struct` Elemento conterá todos os campos comuns a Objetos e Lugares, um campo especial para indicar qual tipo se trata e uma `union` com os dados específicos de cada tipo.

### 4.2.1 Elementos

A `struct` Elemento deve conter os seguintes campos:

- `nome` — nome padrão do Elemento (*string*).
- `curta` — descrição curta (*string*).
- `longa` — descrição detalhada (*string*).
- `ativo` — indicador se o Elemento está ativo no jogo (*char* ou `short int`)[2]. Um Elemento que não está ativo ainda não foi construído ou já foi destruído ou consumido. Em outras palavras, é um indicador se o Elemento faz parte do jogo no momento.
- `visivel` — indicador se o Elemento está visivel. Para o aventureiro conseguir ver o Elemento, ele precisa estar presente e visível.
- `conhecido` — indicador se o Elemento já é conhecido do aventureiro, por exemplo se é um lugar que ele já tenha passado.
- `conteudo` — lista de Objetos contidos no Elemento, pode ser vazia. Por exemplo, Objetos dentro de um lugar, ou dentro de uma bolsa.

[2]Se preferir, defina um tipo booleano com enum

- `acoes` — lista de funções especiais para este Elemento, veja a explicação em aula. É uma lista para ponteiros de funções, faça as adaptações necessárias na definição de lista. Se preferir, pode usar uma tabela de símbolos.
- `animacao` — um ponteiro para função (pode ser `null` que será chamada ao final de cada iteração.
- `detalhe` — a `union` que conterá os campos específicos de Lugar e Objeto.

**Objetos** Os Objetos tem um campo adicional que pode ser uma tabela de símbolos ou uma lista, com atributos adicionais. Cada atributo é um nome associado a um valor que pode ser uma *string* ou um número (use uma `union` para o valor).

Exemplos de atributos são um indicador se o Objeto está vivo, se está quebrado, quantidade de energia (se for uma pilha), quantidade de líquido (se for uma garrafa), etc.

**Lugares** Os lugares terão uma lista de saídas, que são ponteiros para outros Lugares. Algumas saídas são padronizadas e podem ser colocadas como campos: `norte`, `sul`, `leste`, `oeste`. Estes campos podem conter um ponteiro `null` caso não existam.

## 4.3 Comandos

Os comandos correspondem a verbos usados pelo aventureiro. No momento, os comandos serão funções em C. Estas funções têm o seguinte protótipo:

```
int (*func)(*Elemento e1, *Elemento e2);
```

Tanto `e1` como `e2` podem ser `null`, a função deve verificar. Com isso, podemos implementar verbos intransitivos, transitivos e bitransitivos. Veja alguns exemplos de uso:

```
⋮
/* pegue o bolinho:
   retira do local atual e coloca na lista de conteúdo do
   aventureiro
 */
res = pegue(&bolinho, null);

/* chore */
res = chore(null, null);

/* atire o pau no gato */
res = atire(&pau, &gato);

/* res é um indicador de sucesso */
```

## 4.4 Inicialização

A inicialização consiste em montar o mundo onde a aventura acontece. Ela é feita dentro do programa principal, simplesmente criando os Elementos, inicializando e incluindo na tabela de símbolos do jogo.

Além disso, uma variável conterá o Lugar atual do aventureiro. O aventureiro pode ser representado como um Objeto especial mas, se quiser, crie um tipo novo para ele.

Depois da inicialização das variáveis, o programa deve conectar as salas e colocar os objetos em suas posições de início. Dependendo do jogo, esta colocação pode ser aleatória, mas deixemos isso com uma extensão.

O aventureiro pode ter uma inicialização separada, se for mais conveniente.

## 4.5 Esboço do jogo

O jogo começa depois desta preparação e segue o seguinte padrão:

1. Faz uma apresentação inicial, introduzindo a estória e eventuais créditos. A introdução não foi colocada na primeira fase, mas é fácil incluir.

2. Cria um vetor com os comandos de teste e depois disso, temos um laço que executa o seguinte pseudo-código:

   (a) Apresenta a descrição da sala, se ela ainda não foi visitada. Use um atributo para indicar a visita.

   (b) Relaciona o conteúdo visível da sala.

   (c) Manda executar cada um dos comandos, que atualizará os objetos e salas de acordo.

   (d) Faz as atualizações automáticas dos Elementos animados (isto é, que contenham uma função de animação).

## 4.6 O que entregar

Você deve entregar uma versão atualizada dos arquivos da primeira fase.

O programa desta versão deve satisfazer as seguintes condições:

1. Incluir uma descrição inicial de uma estória. Pode ser simples e não precisa ser a estória final completa.

2. Incluir a descrição da sala a cada comando. Na primeira vez deve ser apresentada a descrição longa. Se a sala já foi visitada, apresenta-se a descrição curta.

3. Ao final da descrição longa da sala, imprimir a lista dos objetos incluídos nela, precedidos com artigos indefinidos correspondentes.

4. Faça uma lista dos objetos e salas animadas e ao final de cada iteração execute os comandos correspondentes. Darei mais dicas em aula.

5. Coloque novos objetos e pelo menos 5 salas. Teste todos os verbos e objetos.

6. Coloque verbos específicos em alguns objeto e em algumas salas.

7. Divirta-se!

# 5 Terceira Fase: Interpretação

## 5.1 Comandos do usuário

Esta é aparentemente a parte mais complicada, mas não é um bicho de sete cabeças. O que deve ser feito é ler uma *string* do usuário e traduzí-la em chamadas do programa. Com esta fase pronta deverá ser possível jogar.

Para conseguir entender o comando, usaremos `flex` e `bison`. Embora a gramática não seja tão complicada e possa ser implementada diretamente, o `bison` tornará mais fácil estender e organizar o código.

### 5.1.1 Ponto de partida

A nova versão dos arquivos contém um exemplo relativamente completo de uma micro aventura. Em princípio, basta adaptar estes programas ao que já foi feito. Com isso, o jogo pode ser criado e jogado, completanto o projeto!!!!

O exemplo não usa as funções que foram construídas nas fases anteriores. As tabelas de símbolos, por exemplo, foram adptadas da versão da calculadora, bem como parte das inicializações. As implementações de `Elemento` e `Objeto` também são mais simples.

Você deve usar este exemplo como referência para fazer a implementação de acordo com o esepcificado neste projeto. Como bônus, coloquei a receita para incluir a biblioteca `readline` com o `flex`. Compile e rode o programa para entender o que ela faz.

Vejamos em seguida uma revisão dos arquivos que fazem parte do projeto e a descrição abreviada dos novos.

**main.c** Programa principal. Note o laço vazio na chamada de `yyparse`. O corpo do laço pode conter a chamada para as animações e eventual outro código que precise ser executado a cada passo (imprimir o *status*, etc).

**aventl.l** Analisador léxico, como outros exemplos que vimos para o `flex`. Há algumas coisas a mais, que devem ser destacadas:

Em primeiro lugar, a preparaçao para *readline*. Isto inclui da redefinição do macro `YY_INPUT`. Este macro representa como o `flex` lê a entrada. Neste caso, mudamos a leitura para uma função que usa diretamente a `readline`. Peguei este exemplo da *internet*.

A opção `%caseless` trata maiúsculas e minúsculas como iguais.

Um outro problema é o uso de Unicode. `Flex` não trabalha com unicode, mas trata toso os arquivos como binário. Como em UTF-8 alguns caracteres ocupam mais de um *byte*, é preciso evitar classes nas expressões regulares nestes casos. Veja o código.

**avent.y** Analisador sintático. Aqui está o trabalho pesado do reconhecimento de texto.

Note um macro para facilitar a escrita do código e a função `AcertaF`, que busca a versão correta da função a ser chamada.

Brinque com a gramática, dá para melhorar muitas coisas.

**symrec.c e symrec.h** A tabela de símbolos usada na calculadora, com algumas adaptações. Troque pelo material da primeira fase.

**coisas.c e coisas.h** Este módulo contém a estória e toda a inicialização dos elementos.

Há duas variáveis globais: `inventario`, uma tabela de símbolos com as coisas que o aventureiro carrega, e `Posic`, o lugar onde o aventureiro se encontra.

As funções associadas aos verbos são simplistas, estude para fazer as suas. Não se esqueça de verificar todas as condições.

Troque pelo material da segunda fase, completando e adaptando o que for necessário.

**Makefile** Nada especial, apenas uma regra a mais para gerar a distribuição...

## 5.2 O que entregar

Você deverá fazer todas as adaptações para que seu jogo funcione. Fiz algumas mudanças que podem ser desfeitas se for o caso. Por exemplo a separação entre `Motor` e `Jogo` e a criação do aventureiro como objeto separado. Tome apenas cuidado com o código de serialização.

As modificações devem ser as seguintes, mas se você tiver alguma ideia interessante, pode incluir também, contando bônus:

- Altere o código de serialização para colocar no início do arquivo um código de versão, de modo que o programa recuse ler arquivos de versões diferentes. Se um arquivo de versão errada for carregado, o sistema todo pode ficar inconsistente.
- Inclua tratamento de erros em todos os lugares necessários.
- Aumente a descrição para permitir um adjetivo opcional depois do **objeto**.
- Como exercício adicional, inclua um comando que pergunta se uma palavra existe. Caso ela exista, o programa deve responder que tipo de palavra é, dentre verbo, substantivo ou adjetivo.

A data de entrega será discutida em aula.