Listas con registros, registros con listas y otras combinaciones

Francisco Soulignac

Tecnicatura en Programación Informática
Universidad Nacional de Quilmes

Introducción a la Programación
Algoritmos y Programación

¿Qué hacemos hoy?

1. Repaso
   - Listas
   - Registros
   - Alias de tipo
   - Invariante de representación
2. Recomendaciones al mezclar listas y registros
   - Registros con registros
   - Registros con listas
   - Listas de registros
   - Listas de listas
   - Resumen
3. Ejercicios

Listas

- Representa una colección (finita) ordenada de elementos de un mismo tipo.
- Tipo: List⟨ T ⟩, para T cualquier tipo.
  - List⟨ int ⟩, List⟨ Dir ⟩, List⟨ List⟨ int ⟩ ⟩, List⟨ Color ⟩
- Valores (en papel): $[x_1, \ldots, x_n]_T$ con x_i elementos de tipo T.
- Generadores de valores (en CGobstones): Nil⟨ T ⟩(), Cons(x, xs).
  - Nil⟨ int ⟩(), Cons(1, Nil⟨ int ⟩()), Cons(Norte, Cons(Norte), Nil⟨ Dir ⟩())
- Otras expresiones: isNil, Snoc, head, last, tail, init.

Operaciones de Listas

Operación	Descripción	Ejemplo/s (signo = es "denota")
Nil$\langle\ T\ \rangle()$:	Denota la lista vacía cuyos elementos tienen tipo T	Nil$\langle$ int $\rangle() = []$ Nil$\langle$ List$\langle$ int $\rangle\ \rangle() = []$
Cons(x, xs):	Denota la lista que se obtiene de agregar x al inicio de L	Cons$(x, [x_1, \ldots, x_n]) =$ $[x, x_1, \ldots, x_n]$
isNil(xs):	Denota `true` si y sólo si xs denota una lista vacía	isNil$([]) =$ true isNil$([x_1, \ldots, x_n]) =$ false
Snoc(xs, x):	Denota la lista que se obtiene de agregar x al final de L	Snoc$([x_1, \ldots, x_n], x) =$ $[x_1, \ldots, x_n, x]$
head(xs):	Denota el primer elemento de xs Precondición: `not` isNil(xs)	head$([x_1, \ldots, x_n]) = x_1$
last(xs):	Denota el último elemento de xs Precondición: `not` isNil(xs)	last$([x_1, \ldots, x_n]) = x_n$
tail(xs):	Denota la lista que se obtiene de eliminar el primer elemento de xs. Precondición: `not` isNil(xs)	tail$([x_1, \ldots, x_n]) = [x_2, \ldots, x_n]$
init(xs)	Denota la lista que se obtiene de eliminar el último elemento de xs. Precondición: `not` isNil(xs)	init$([x_1, \ldots, x_n]) = [x_1, \ldots, x_{n-1}]$

Registros

- Representa una agrupación de datos, no necesariamente del mismo tipo, que representan un tipo nuevo.
- Tipo: `struct` T $\{\langle$ campos $\rangle\}$ T; define un nuevo tipo de datos T.
- $\langle$ campos $\rangle$ contiene la declaración (tipo + identificador) de los campos de T.

Ejemplos:

- `struct` Coordenada{int fila; int columna;};
- `struct` Perro{int edad; Color pelo; Color ojos;};
- `struct` Alumno{int legajo; int cursadas; int aprobadas;};
- `struct` Barco{Coordenada pos; Dir orientacion; int tamanho; List$\langle$ bool $\rangle$ hundido;};
- `struct` Jugador{List$\langle$ Barco $\rangle$ barcos; Color ficha;};

Registros: declaración de variables y acceso a campos

- Las variables de un tipo registro se declaran igual que las variables de tipos básicos.
  - Coordenada c; Barco perlaNegra; Jugador barbaRoja;
- Inicialmente, cada campo tiene el valor basura.
- El operador . se utiliza para acceder a los campos de un tipo.
- Los campos son expresiones del tipo correspondiente; i.e., denotan un valor.
  - **if**(fran.cursadas > fran.aprobadas * 2) . . .;
  - **if**(isNil(barbaRoja.barcos)) . . .;
  - head(barbaRoja.barcos);
  - last(head(barbaRoja.barcos).hundido);
  - bool perdio(Jugador j);
  - int barcosHundidos(Jugador j);
  - Barco elOrgulloDeLaFlota(Jugador j);
- Para una variable de registro, sus campos son variables; i.e., se pueden asignar
  - c.fila = 0; garabato.edad = 4;
  - perlaNegra.pos.fila = 0;

Registros: inicialización

- Inicialmente, cada campo tiene el valor basura.
- Como convención, definimos procedimientos de inicialización con prefijo `mk`.
- Estos procedimientos emulan el concepto de valor.
  - Coordenada mkCoordenada(int x, int y) {...}
  - Perro mkPerro(int edad, Color pelo, Color ojos) {...}
  - Barco mkBarco(Coordenada pos, Dir orientacion, int tamanho, List⟨ Bool ⟩ hundido) {...}
  - Jugador mkJugador(List⟨ Barco ⟩ barcos, Color ficha) {...}
- A veces, algunos valores se pueden inicializar con datos por defecto.
  - Alumno mkIngresante(int legajo) {
    Alumno ret;
    ret.legajo = legajo; ret.cursadas = 0; ret.aprobadas = 0;
    return ret;
    }

Alias de tipo

- El comando `typedef` permite dar un nuevo nombre a un tipo.
- Se escribe `typedef` TipoExistente TipoNuevo;
  - `typedef` Coordenada Casilla;
  - `typedef` List⟨ Celda ⟩ Fila;
  - `typedef` List⟨ Dir ⟩ Camino;

- Muy útil cuando hay que usar listas dentro de registros.
  - `typedef` List⟨ Barco ⟩ Flota;
    `struct` Jugador{Flota barcos; Color ficha;};

Invariante de representación

- Condiciones que debe cumplir una estructura de datos para reflejar de manera fiel la realidad que pretende modelar.
  - `typedef struct` {int legajo, cursadas, aprobadas;} Alumno;
    Inv. Rep.: legajo $>$ 0 && cursadas $\geq$ aprobadas $\geq$ 0.
  - `typedef struct` {Coordenada pos; Dir orientacion; int tamanho; List$\langle$ Bool $\rangle$ hundido;} Barco;
    Inv. Rep.: hundido tiene tamanho cantidad de elementos,
    pos.fila $>$ 0 && pos.columna $>$ 0,
    la distancia de pos al borde del tablero, recorriendo en direccion orientación, es al menos tamanho.
  - `typedef` List$\langle$ Barco $\rangle$ Flota;
    Inv. Rep.: No hay dos barcos cuyos dibujos compartan coordenadas.

Invariante de representación

- Condiciones que debe cumplir una estructura de datos para reflejar de manera fiel la realidad que pretende modelar.
  - `typedef struct` {int legajo, cursadas, aprobadas;} Alumno;
    Inv. Rep.: legajo $>$ 0 && cursadas $\geq$ aprobadas $\geq$ 0.
  - `typedef struct` {Coordenada pos; Dir orientacion; int tamanho; List$\langle$ Bool $\rangle$ hundido;} Barco;
    Inv. Rep.: hundido tiene tamanho cantidad de elementos,
    pos.fila $>$ 0 && pos.columna $>$ 0,
    la distancia de pos al borde del tablero, recorriendo en direccion orientación, es al menos tamanho.
  - `typedef` List$\langle$ Barco $\rangle$ Flota;
    Inv. Rep.: No hay dos barcos cuyos dibujos compartan coordenadas.

- Podemos suponer que el invariante se satisface para cada parámetro (salvo indicación en contrario).
- Debemos garantizar que el valor de retorno de un procedimiento satisface el invariante (salvo indicación en contrario).

Registros con registros

- Regla de los dos puntos: evitar expresiones del tipo
  `r.campo_de_tipo_registro.campo_interno`
  Sobretodo para asignaciones.
- Usar funciones que accedan/modifiquen el registro interno.
- De esta forma, evitamos errores y dividimos mejor las subtareas.

Registros con registros

- Regla de los dos puntos: evitar expresiones del tipo
  `r.campo_de_tipo_registro.campo_interno`
  Sobretodo para asignaciones.
- Usar funciones que accedan/modifiquen el registro interno.
- De esta forma, evitamos errores y dividimos mejor las subtareas.

```
struct Desplazamiento{int norte; int este;};

Barco DesplazarBarco(Barco barco, Desplazamiento desplazamiento) {
1. Barco res = barco;
2. res.pos.fila = res.pos.fila + desplazamiento.norte; //NO
3. res.pos.columna = res.pos.columna + desplazamiento.este; //NO
4. return res;
}
```

Registros con registros

- Regla de los dos puntos: evitar expresiones del tipo
  `r.campo_de_tipo_registro.campo_interno`
  Sobretodo para asignaciones.
- Usar funciones que accedan/modifiquen el registro interno.
- De esta forma, evitamos errores y dividimos mejor las subtareas.

```
struct Desplazamiento{int norte; int este;};

Barco DesplazarBarco(Barco barco, Desplazamiento desplazamiento) {
1. Barco res = barco;
2. res.pos = DesplazarCoordenada(res.pos, desplazamiento); //SI
3. return res;
}
```

Registros con listas

- Evitar multiples asignaciones un campo de tipo lista dentro de una función.
  - Mejor asignar una unica vez el resultado de una función.
- En particular, evitar recorridos sobre campos de tipo lista.
  - Mejor invocar una función que realice el recorrido.
- Recordar: `Cons` y `Snoc` no tienen efectos.

Registros con listas

- Evitar multiples asignaciones un campo de tipo lista dentro de una función.
  - Mejor asignar una unica vez el resultado de una función.
- En particular, evitar recorridos sobre campos de tipo lista.
  - Mejor invocar una función que realice el recorrido.
- Recordar: `Cons` y `Snoc` no tienen efectos.

```
Jugador QuitarBarcosHundidos(Jugador j) {
1. Jugador recorrido = j; Jugador resultado = j;
2. resultado.flota = Nil⟨Barco⟩(); //RARO
3. while(not is Nil(recorrido.flota)) {
4.         if(not hundido(head(recorrido.flota))) {
5.                 resultado.flota = Snoc(resultado.flota, head(recorrido.flota));
6.         }
7.         recorrido.flota = tail(recorrido.flota);
8. }
9. return resultado;
}
```

Mucha mezcla de funciones con operador punto.

Registros con listas

- Evitar multiples asignaciones un campo de tipo lista dentro de una función.
  - Mejor asignar una unica vez el resultado de una función.
- En particular, evitar recorridos sobre campos de tipo lista.
  - Mejor invocar una función que realice el recorrido.
- Recordar: `Cons` y `Snoc` no tienen efectos.

```
Jugador QuitarBarcosHundidos(Jugador j) {
1. Jugador resultado = j;
2. resultado.flota = QuitarHundidosDeFlota(j.flota); //MEJOR!
3. return resultado;
}

Flota QuitarHundidosDeFlota(Flota f) { //UN FILTRO COMUN Y CORRIENTE
1. Flota recorrido = f; Flota resultado = Nil⟨Barco⟩();
2. while(not is Nil(recorrido)) {
3.         if(not hundido(head(recorrido))) {
4.                 resultado = Snoc(resultado, head(recorrido));
5.         }
6.         recorrido = tail(recorrido);
7. }
8. return resultado;
}
```

Listas de registros

- Evitar la modificación de los registros mientras se recorre.
  - En cambio, separar el recorrido de la modificación de registros.
- Tener cuidado al querer cambiar un campo de un registro de la lista.
  - Recordar: head(lista) **NO** es una variable.
  - En consecuencia, head(lista).campo **NO** es asignable.

Listas de registros

- Evitar la modificación de los registros mientras se recorre.
  - En cambio, separar el recorrido de la modificación de registros.
- Tener cuidado al querer cambiar un campo de un registro de la lista.
  - Recordar: head(lista) **NO** es una variable.
  - En consecuencia, head(lista).campo **NO** es asignable.

```
Flota ReflejarFlota(Flota f) {
1.  Flota recorrido = f; Flota resultado = Nil⟨Barco⟩();
2.  Barco auxiliar; //RARO
3.  while(not is Nil(recorrido)) {
4.          //NO HACER LO COMENTADO: ERROR GARRAFAL!
5.          //head(resultado).orientacion = opuesto(head(resultado).orientacion);
6.          auxiliar = head(resultado);
7.          auxiliar.orientacion = opuesto(auxiliar.orientacion);
8.          resultado = Snoc(resultado, auxiliar);
9.          recorrido = tail(recorrido);
10. }
11. return resultado;
}
```

Mezcla de recorrido con modificacion de registros (UGH).

Listas de registros

- Evitar la modificación de los registros mientras se recorre.
  - En cambio, separar el recorrido de la modificación de registros.
- Tener cuidado al querer cambiar un campo de un registro de la lista.
  - Recordar: head(lista) **NO** es una variable.
  - En consecuencia, head(lista).campo **NO** es asignable.

```
Flota ReflejarFlota(Flota f) { //RECORRIDO COMUN Y CORRIENTE
1. Flota recorrido = f; Flota resultado = Nil⟨Barco⟩();
2. while(not is Nil(recorrido)) {
3.         resultado = Snoc(resultado, ReflejarBarco(head(recorrido)));
4.         recorrido = tail(recorrido);
5. }
6. return resultado;
}

Barco ReflejarBarco(Barco b) { //MODIFICACION COMUN Y CORRIENTE
1. Barco res = b;
2. res.orientacion = opuesto(res.orientacion);
3. return res;
}
```

Listas de Listas

- Evitar el acceso y procesamiento de las listas internas
  - Casi natural en muchos casos, por falta de while anidados, pero . . .
  - . . . a no dormirse

Listas de Listas

- Evitar el acceso y procesamiento de las listas internas
  - Casi natural en muchos casos, por falta de while anidados, pero . . .
  - . . . a no dormirse

```
List⟨ List⟨ int ⟩ ⟩ Separar(List⟨ int ⟩ l, int separador) {
1.  List⟨ int ⟩ recorrido = l; List⟨ int ⟩ auxiliar = Nil⟨ int ⟩(); //RARO
2.  List⟨ List⟨ int ⟩ ⟩ resultado = Nil⟨ List⟨ int ⟩ ⟩();
3.  while(not is Nil(recorrido)) {
4.          if(head(recorrido) != separador) {
5.                  auxiliar = Snoc(auxiliar, head(recorrido));
6.          } else {
7.                  resultado = Snoc(resultado, auxiliar);
8.                  auxiliar = Nil⟨ int ⟩();
9.          }
10.         recorrido = tail(recorrido);
11. }
12. return Snoc(resultado, auxiliar);
}
```

Listas de Listas

- Evitar el acceso y procesamiento de las listas internas
  - Casi natural en muchos casos, por falta de while anidados, pero . . .
  - . . . a no dormirse

```
List⟨ List⟨ int ⟩ ⟩ Separar(List⟨ int ⟩ l, int separador) {
1. List⟨ int ⟩ recorrido = l;
2. List⟨ List⟨ int ⟩ ⟩ resultado = Nil⟨ List⟨ int ⟩ ⟩();
3. while(not is Nil(recorrido) && pertenece(recorrido, separador)) {
4.         resultado = concatenacion(resultado, primeros(recorrido,
   posicion(separador)));
5.         recorrido = sinPrimeros(recorrido, posicion(separador));
6. }
7. return Snoc(resultado, recorrido);
}

int posicion(List⟨ int ⟩, int valor) {. . . }
List⟨ int ⟩ primeros(List⟨ int ⟩, int cantidad) {. . . }
List⟨ int ⟩ sinPrimeros(List⟨ int ⟩, int cantidad) {. . . }
```

Resumen de recomendaciones

- Separar el procesamiento de registros y listas internas.
- Usar funciones para resolver los problemas sobre los elementos internos.
- De esta forma, cada función opera sobre una lista o sobre un registro, tal cual lo vimos en las clases anteriores.

Resumen de recomendaciones

- Separar el procesamiento de registros y listas internas.
- Usar funciones para resolver los problemas sobre los elementos internos.
- De esta forma, cada función opera sobre una lista o sobre un registro, tal cual lo vimos en las clases anteriores.

- Como ventaja adicional, obtenemos funciones reutilizables para procesar los elementos internos.

Ejercicios