Documentazione progetto Java

Introduzione

È stata realizzata una piccola applicazione in Java per gestire una riserva di caccia. L'obbiettivo principale dell'applicazione è di salvare e gestire i dati riguardanti gli animali nella riserva, i cacciatori e gli accessori in vendita. Il focus principale non è stato realizzare un'applicazione completa e funzionale, ma piuttosto utilizzare i diversi costrutti messi a disposizione da Java.

Struttura

L'applicazione è composta da 4 packages:

- **animali**: contiene la classe astratta Animale, estesa da tutte le classi concrete di animali, che in questo caso sono Cervo, Quaglia e Orso. Ogni Animale ha un id univoco generato in automatico e altri campi come peso ed età. Le tre classi che estendono Animale hanno campi aggiuntivi come il peso del palco per Cervo e un booleano per indicare il letargo per Orso. L'interfaccia Cacciabile, implementata solo da Cervo e Quaglia, contiene solo un metodo, invocabile appunto solo dagli animali della riserva che possono essere cacciati. Infine, il package contiene l'enumerativo Genere (MASCHIO o FEMMINA) e la classe ComparatorPeso che implementa Comparator<Animale> per confrontare gli animali in base al loro peso.
- **cacciatori**: contiene la classe Cacciatore. Ogni cacciatore ha (come per gli animali) un id univoco generato in automatico. Inoltre, ha un campo denominato trofei, che rappresenta una lista di animali cacciabili che quel cacciatore ha cacciato nella riserva. Nel package c'è anche la classe ComparatorPunti per confrontare i cacciatori in base al loro punteggio.
- **riserva**: Contiene la classe principale, ovvero Riserva. Questa rappresenta la riserva di caccia, ha tre liste: una di animali, una di cacciatori e l'ultima di accessori (Accessorio è un record, anch'esso presente in questo package). Contiene poi i metodi per modificare e consultare le liste:
  - un metodo per ogni lista che restituisce una stringa che descrive ogni elemento della lista;
  - un metodo per ogni lista per aggiungere nuovi elementi alle liste;
  - il metodo uccidi che preso l'id di un cacciatore e di un animale aggiunge l'animale tra i trofei del cacciatore, segna l'animale come morto e aggiunge al cacciatore i punti. L'animale può essere cacciato solo se è un'istanza dell'interfaccia Cacciabile;
  - Il metodo cacciatoriMigliori che restituisce i tre cacciatori con più punti;
  - Il metodo animaliCacciabili che restituisce la lista degli animali attualmente disponibili per la caccia nella riserva;
  - Il metodo animalePiuPesante che restituisce l'animale più pesante (ancora vivo) nella riserva.

  Infine, il package contiene tre classi che estendono Exception utili a modellare diversi comportamenti indesiderati nell'applicazione.
- **test**: Contiene solo la classe TestRiserva con il main, serve a testare le principali funzionalità dell'applicazione.

Caratteristiche

Segue una lista di caratteristiche e costrutti tipici di Java utilizzati per implementare l'applicazione, con riferimento al punto del codice dove sono stati utilizzati.

- **Ereditarietà**: concetto fondamentale della programmazione ad oggetti. In questo caso l'ereditarietà è stata utilizzata per rappresentare le varie specie di animali: tutte sono classi che hanno in comune un certo numero di campi e metodi raggruppati nella super-classe astratta Animale. Questa classe è astratta perché non può essere istanziata (visto che non esistono animali "generici" senza specie). Ovviamente è stato sfruttato anche il concetto di **overriding** per i metodi delle sottoclassi (come per il metodo toString).
- **Interfacce**: è stata definita l'interfaccia Cacciabile così da poter definire quali animali possono essere cacciati e quali no. L'interfaccia ha il solo metodo cacciato, che viene implementato in maniera differente dalle varie sottoclassi di Cacciabile, in generale però il metodo imposta l'animale come morto, lo aggiunge alla lista di "premi" del cacciatore (il quale è passato come parametro) e aumenta il numero di punti del cacciatore. Nella lista di "premi" in Cacciatore gli elementi sono appunto di tipo Cacciabile, viene così sfruttato il concetto di **sottotipazione**. In maniera analoga la lista di animali della riserva è una lista di elementi della classe Animale, ma può contenere (e di fatto contiene) oggetti il cui tipo effettivo è un sottotipo di Animale.
- **Overload**: è stato fatto overload dei costruttori della classe Animale e delle relative sottoclassi.
- **Visibilità**: i campi delle classi hanno visibilità differenti:
    - I campi delle classi nel package animali sono protected, ovvero visibili solo nel package e nelle sottoclassi;
    - I campi nella classe Cacciatore hanno la visibilità di default, ovvero sono visibili solo nel package di appartenenza;
    - I campi nella classe Riserva sono privati, cioè visibili solo all'interno della classe scelta.

  I metodi sono invece public, ovvero accessibili ovunque. Queste distinzioni sono utili per proteggere le informazioni delle classi, in questo progetto sono stati utilizzati tutti più per dimostrazione che non per necessità.

- **Static**: Un componente statico è condiviso da tutte le istanze della classe. Nelle classi Animale e Cacciatore è presente un campo statico count, il cui scopo è quello di contare il numero di istanze della classe così da poter assegnare ad ogni istanza un id unico. Inoltre, la classe Riserva implementa il design pattern **Singleton** (così che si possa creare al più una sola istanza della classe), di conseguenza ha un campo statico di tipo Riserva (che rappresenta l'unica istanza della classe) e un metodo statico pubblico che crea l'istanza chiamando il costruttore privato (se non è ancora stata creata in precedenza) e la ritorna.
- **Java Collection Framework**: tutte le strutture dati utilizzate sono delle istanze della classe generica ArrayList<T> del JCF. È stato quindi sfruttato il potenziale della programmazione generica.
- **Stream**: sugli oggetti che estendono Collection<T> del JCF (tra cui ArrayLisy<T>) si può invocare il metodo stream che permette di operare sulle collezioni in maniera "funzionale" ovvero applicando una serie di funzioni ai dati della collezione. Per esempio, in questo progetto gli stream sono stati utilizzati per ottenere la lista dei tre migliori cacciatori della riserva a partire dalla lista completa, per farlo è stato necessario ordinare gli elementi della lista (ed usare di conseguenza un algoritmo di sorting) in base al campo punti (usando quindi anche un'istanza della classe ComparatorPunti).
- **Lambda expressions**: sono blocchi di codice che prendono come input un parametro e ritornano un valore. La particolarità è che possono essere passate come parametri dei metodi ed essere implementate direttamente all'interno della chiamata. In questo progetto sono state utilizzate con gli stream, in particolare come parametri del metodo filter per specificare quali elementi della lista scartare e quali mantenere nel risultato finale dello stream. I metodi animaliCacciabili e

animalePiuPesante di Riserva sono stati implementati utilizzando sia gli stream che le lambda expression.

- **Final**: Le classi Riserva e Cacciatore sono final, ovvero non è possibile che altre classi le estendano. Alcuni campi sono indicati come final perché una volta impostatone il valore non possono più essere modificati, ad esempio i campi id di Animale e Cacciatore sono contrassegnati come final. Infine, nel metodo addTrofeo di Cacciatore il parametro di tipo Cacciabile è indicato come final, di conseguenza all'interno del metodo non sarà possibile modificarlo.
- **var**: questa keyword indica che il tipo della variabile a cui è applicato non è specificato, ma sarà dedotto in automatico dal compilatore. In questo progetto è stato utilizzato nel metodo animalePiuPesante semplicemente per evitare di indicare il tipo di una variabile (ovvero Optional<Animale>).
- **Record**: Sono delle classi con campi privati immutabili, non è necessario implementare il costruttore, i metodi getter e setter e il metodo toString perché sono già presenti in automatico. Nel progetto è stato usato il record Accessorio per rappresentare oggetti messi in vendita nella riserva.
- **Enum**: è una speciale tipologia di "classe" che rappresenta un insieme di costanti. Nel progetto è stato implementato il tipo enumerativo Genere che contiene due possibili valori: MASCHIO e FEMMINA.
- **Varargs**: I Variable Arguments permettono di passare ad un metodo un numero indefinito di parametri (dello stesso tipo), l'utilizzo dei varargs non preclude di passare altri parametri in modo convenzionale, le uniche imposizioni sono che può esserci al più un varargs per metodo e deve essere l'ultimo tra i parametri. Nella classe Riserva i tre metodi che aggiungono elementi alle liste usano i varargs così da poter aggiungere quanti elementi si vuole chiamando una sola volta il metodo.
- **Optional**: La classe generica Optional<T> rappresenta un "contenitore" di una variabile di tipo T, è utile per gestire riferimenti a null senza scontrarsi con la NullPointerException. Fornisce dei metodi per verificare se l'oggetto vero e proprio è presente o per ritornare un Optional vuoto. Nella classse Riserva il metodo animalePiuPesante ritorna un Optional<Animale>, se la lista di animali è vuota (e quindi non c'è un animale più pesante degli altri) viene ritornato Optional.empty().

Documentazione progetto C++

Introduzione

L'applicazione realizzata in C++ ha il medesimo obiettivo di base di quella in Java, ovvero gestire i dati di una riserva di caccia. Le funzionalità non sono però le stesse, sono state apportate diverse modifiche per poter sfruttare le caratteristiche di C++. Ovviamente, anche in questo caso, il focus principale non è l'applicazione in sé, ma piuttosto l'utilizzo dei vari costrutti C++.

Struttura

L'applicazione è composta da 3 namespace:

- **animali**: contiene la classe astratta Animale simile al corrispettivo in Java solo che l'enumerativo per il genere è stato sostituito con un semplice booleano. Inoltre, si è deciso di rendere tutti gli animali cacciabili; quindi, la classe Animale contiene il metodo astratto e virtuale cacciato. Ci sono due classi che estendono pubblicamente Animale, ovvero Lupo e Coyote. La prima aggiunge ad Animale il metodo ulula (semplicemente per differenziarla un po' dalla classe base), la seconda aggiunge un campo di tipo puntatore a Tana (Tana che è una struct con tre campi). Infine, la classe CoyWolf estende pubblicamente sia Coyote che Lupo (rappresenta un incrocio tra i due animali) e non aggiunge nulla in più a ciò che eredita.
- **cacciatori**: contiene solo la classe Cacciatore, praticamente identica al corrispettivo Java, solo che al posto di una lista di animali cacciabili ha come campo semplicemente una lista di puntatori (smart) ad animali.
- **riserva**: contiene la classe Riserva. Contiene due liste di puntatori smart: una verso elementi della classe Animale e l'altra della classe Cacciatori. Non sono stati implementati gli accessori. Questa classe contiene metodi per aggiungere elementi alle liste, restituire una stringa che rappresenta le liste, registrare l'uccisione di un animale da parte di un cacciatore e restituire la lista dei tre cacciatori migliori.

Infine, il file TestRiserva.cpp contiene il metodo main e serve appunto a testare le principali funzionalità dell'applicazione.

Caratteristiche

Segue una lista di caratteristiche e costrutti tipici di C++ utilizzati per implementare l'applicazione, con riferimento al punto del codice dove sono stati utilizzati.

- **Classi**: C++ permette di sfruttare il paradigma di programmazione ad oggetti, ma a differenza di Java supporta anche la programmazione procedurale ("ereditata" dal C). In ogni caso il progetto è stato realizzato sfruttando principalmente la programmazione ad oggetti e quindi il concetto di classe. Solo due funzioni sono all'esterno delle classi, ovvero il main e comparatorByPunti di cui si parlerà in seguito.
- **File .h e .cpp**: le classi Animale, Cacciatore e Riserva sono implementate in due file ciascuna, con lo stesso nome della classe e con estensione .h e .cpp. Nel file header (ovvero il .h) è presente la definizione della classe che contiene i campi e la dichiarazione dei metodi. L'implementazione dei metodi è nel corrispettivo file .cpp. Quindi il file header contiene l'interfaccia della classe (ed è quello che viene importato negli altri file all'occorrenza), mentre il .cpp contiene l'implementazione della

logica della classe. Invece le classi Lupo, Coyote e CoyWolf sono interamente contenute in un file .h ciascuno che contiene sia la classe con la definizione dei metodi che l'implementazione degli stessi all'esterno della classe. Il file Coyote.h contiene anche la definizione della struct Tana. Infine, il main è contenuto nel file TestRiserva.cpp.

- **Visibilità**: In C++ all'interno delle classi si possono specificare tre diverse visibilità:
  - **private**: per i membri visibili solo all'interno della classe. Sono tali i campi di Cacciatore e Riserva e il campo statico count di Animale.
  - **protected**: per i membri visibili all'interno della classe o di una classe derivata. Sono tali i campi di Animale e di Coyote (e di conseguenza delle relative classi derivate, essendo l'ereditarietà utilizzata pubblica).
  - **public**: per i membri visibili ovunque. Hanno questa visibilità tutti i metodi di tutte le classi.
- **Ereditarietà**: In C++ l'ereditarietà è più flessibile che in Java. È infatti possibile che una classe erediti il codice di più classi base. Inoltre, si possono avere diversi **tipi di ereditarietà**: public, protected e private. In tutti i casi la classe derivata eredita tutto il codice della classe base, ma se l'ereditarietà è private o protected la visibilità dei membri ereditati viene ristretta. In quest'ultimo caso la classe derivata non è **sottotipo** della classe base. L'ereditarietà pubblica è invece come quella Java, la visibilità dei membri ereditati non cambia e quindi la classe derivata è sottotipo della classe base. In C++ questo è l'unico modo per avere sottotipazione. In questa applicazione è presente la seguente struttura: entrambe le classi Lupo e Coyote ereditano pubblicamente da Animale, la classe CoyWolf eredità pubblicamente sia da Lupo che da Coyote. Di conseguenza tutte le classi derivate sono sottotipo di Animale, inoltre CoyWolf è sottotpo sia di Lupo che di Coyote. Questa particolare struttura di ereditarietà solleva un problema: le istanze della classe CoyWolf ereditano il codice della classe Animale due volte, una volta per ciascuna classe "intermedia" (ovvero Lupo e Coyote), quindi per esempio un'istanza di CoyWolf avrebbe due campi peso, ovvero Lupo::Animale::peso e Coyote::Animale::Peso. Per evitare ciò si costruisce un **diamante**: si indica che Lupo e Coyote ereditano da Animale in maniera virtuale, così le classi che erediteranno da Lupo e Coyote, erediteranno da Animale soltanto una volta, come se l'istanza di animale ereditata fosse condivisa dalle due istanze di Lupo e Coyote a loro volta ereditate dall'istanza di CoyWolf.
- **Costruttori**: In tutti i costruttori di questo progetto viene sfruttata la **lista di inizializzazione** che permette di inizializzare i campi della classe in maniera concisa ed elegante. Ovviamente i costruttori delle classi derivate contengono la chiamata al costruttore della classe base.
- **Distruttori**: Le classi Riserva e Cacciatore non definiscono esplicitamente un distruttore perché non è necessario che facciano operazioni di delete o altro; quindi, si sfrutta il distruttore di default. Le classi che sfruttano il meccanismo di ereditarietà hanno invece definito un distruttore virtuale (quello di default non è definito virtuale), in modo che quando una classe derivata viene distrutta venga effettuata anche la chiamata al distruttore della classe base. L'unico distruttore non vuoto è quello di Coyote, che avendo un campo che è un puntatore ad un oggetto nello heap (allocato nel costruttore) deve fare il corrispettivo delete nel distruttore.
- **Overriding**: Se una classe base definisce un metodo come **virtual**, le classi derivate possono fare override del metodo. In questo caso (e se si lavora con variabili puntatori) si sfrutta il binding dinamico, ovvero il metodo (virtual) che viene eseguito dipende non dal tipo della variabile ma da quello dell'oggetto vero e proprio a runtime. I metodi virtual nell'applicazione sono toString e cacciato di Animale, di cui tutte le classi derivate fanno l'override.
- **Classi astratte**: La classe Animale è astratta in quanto definisce il metodo virtuale cacciato che a sua volta è astratto. Ciò significa che la classe non fornisce l'implementazione del metodo che viene

delegata alle classi derivate (che ne devono fare l'override). Inoltre, ciò implica che la classe Animale non possa essere istanziata.

- **Passaggi di parametri**: In C++ si possono passare i parametri delle funzioni in tre modi:
  - **Passaggio per valore**: viene passato il contenuto della variabile; quindi, nella funzione posso operare su un parametro che ha lo stesso valore della variabile passata ma che non è collegato ad essa: se il parametro viene modificato la variabile originaria rimane invariata. Nell'applicazione, ad esempio, nel metodo setVivo(bool vivo) di Animale il booleano è passato per valore.
  - **Passaggio per puntatore**: viene passato un puntatore (ovvero un indirizzo di memoria) alla variabile che si vuole passare. I vantaggi sono molteplici: la dimensione del puntatore è solitamente più piccola di quella della variabile puntata (in particolar se l'elemento puntato è un oggetto di notevoli dimensioni) e permette di modificare la variabile originale all'interno della funzione. Nell'applicazione i metodi addAnimale e addCacciatore di riserva hanno come parametri un puntatore (smart) rispettivamente ad un Animale e ad un Cacciatore.
  - **Passaggio per riferimento**: il principio è uguale al passaggio per puntatore, solo che cambia la sintassi: colui che chiama la funzione passa la variabile come nel passaggio per valore, la funzione stessa potrà operare sul parametro ottenuto proprio come se fosse la variabile originaria. È come se si assegnasse ed utilizzasse un alias della variabile originale. Il funzionamento pratico è uguale a quello dei puntatori. Nell'applicazione, ad esempio, il metodo setNome di Cacciatore prende come parametro un riferimento ad una stringa (costante) che viene assegnato al campo nome della classe.
- **default argument**: In una funzione si può indicare il valore di default che un parametro assume nel caso in cui il chiamante non specifichi nulla per tale parametro. I costruttori di Animale e delle relative classi derivate sfruttano questo meccanismo per impostare l'età dell'animale a 0 nel caso in cui non sia stato passato un valore nella chiamata al costruttore.
- **return by reference**: Nel metodo getNome di cacciatore viene ritornato un string&, ovvero un riferimento ad una stringa, che in questo caso altro non è che il riferimento al campo nome della classe. Ciò significa che il chiamante otterrà un alias al campo stesso, che non potrà essere modificato perché indicato come costante.
- **const**: la parola chiave const è stata utilizzata per diversi scopi:
  - il campo id (sia di Animale che di Cacciatore) non può essere modificato perché indicato come const.
  - i metodi getter (come, ad esempio, getEta di Animale) sono indicati come const perché non possono modificare l'istanza della classe.
  - alcuni metodi getter (come quelli relativi a campi costanti, come isMaschio di Animale) oltre che essere const ritornano un elemento a sua volta const, ovvero che non potrà essere modificato dal chiamante.
  - il metodo setNome di Cacciatore ha come parametro un riferimento a string, visto che questo non deve essere modificato dal metodo è passato come const.
- **campi statici**: Le classi Animale e Cacciatore hanno un campo (denominato count) indicato come static, ovvero condiviso tra tutte le istanze della classe. Viene utilizzato per tenere il conto del numero di istanze della classe create e per poter quindi assegnare ad ognuna di esse un semplice id univoco.
- **Funzioni statiche**: nel file Riserva.cpp è presente la funzione comparatorByPunti, esterna alla classe Riserva ed utilizzata solo da un metodo di quest'ultima. Questa classe è static, ovvero ha *internal linkage* e quindi visibilità ridotta.

- **inline**: una funzione inline ha lo stesso comportamento di una non definita come tale, però viene eseguita più velocemente perché quando viene chiamata il codice della funzione viene copiato direttamente in corrispondenza della chiamata (quindi non deve essere creato un record di attivazione apposito nello stack).
- **Overload di <<**: C++ permette di fare l'**overload** degli operatori, come "+", "=" o appunto "<<". Ciò significa che si può ridefinire il comportamento dell'operatore in base al tipo delle variabili su cui è utilizzato. Nell'applicazione è stato fatto l'overload di "<<" con la seguente segnatura:

```
ostream& operator<<(ostream& out, const Cacciatore& cacciatore)
```

  In questo modo si potranno quindi stampare su console le informazioni di un'istanza della classe cacciatore senza chiamare nessun metodo (come è invece necessario fare per Animale, che ha il metodo toString).
- **friend**: una classe può indicare delle funzioni o altre classi come "amiche", esse potranno accedere ad ogni membro della classe, anche a quelli privati. Nell'applicazione la classe Cacciatore indica l'overload dell'operatore "<<" come friend, in modo che possa accedere direttamente ai suoi campi privati.
- **smart pointers**: sono delle classi con lo scopo di "sostituire" l'utilizzo dei puntatori convenzionali. Il vantaggio è che non è necessario esplicitare l'operazione di delete del puntatore perché viene gestito in automatico una volta che lo smart pointer viene deallocato dallo stack. In questo progetto tutte le liste contengono degli shared_ptr<T> (che tra l'altro, come si può notare, sono dei **template**). Sono "condivisi" perché un singolo oggetto nello heap può essere puntato da più shared_ptr, ciascuno dei quali ha un counter del numero di puntatori verso quell'oggetto, quando il conteggio va a 0 viene chiamato il delete in automatico. Gli smart pointers fanno overload di "*" e "->" e possono essere quindi utilizzati come puntatori normali.
- **STL**: La Standard Template Library contiene una serie di **contenitori**, **algoritmi** e **iteratori** generici (ovvero basati sui template). Le liste in questo progetto sono istanze di list<T> della standard template library. Nel metodo cacciatoriMigliori di Riserva viene chiamato il metodo sort sulla lista dei cacciatori, questa applica un algoritmo di ordinamento sulla lista (basato sulla funzione comparatorByPunti passata per parametro). Infine, per scorrere le liste si usano gli iteratori, ovvero oggetti (anch'essi generici) che permettono di visitare ogni elemento della lista.
- **auto**: questa parola chiave, del tutto simile a var di Java, permette di non indicare esplicitamente il tipo di una variabile che viene ricavato in automatico dal compilatore. Nel progetto è utilizzato per evitare di indicare il tipo degli iteratori nei for loop (ad esempio nel metodo cacciatoriMigliori di Riserva)

Documentazione progetto Haskell

La componente principale della piccola applicazione Haskell è la funzione binarySearch che prende in input un algoritmo di sorting, un elemento di tipo ordinabile "a" e una lista di elementi dello stesso tipo, l'output è la seguente tupla: (lista ordinata, elemento da cercare, posizione dell'elemento nella lista ordinata). Praticamente ordina la lista (usando l'algoritmo d sorting passato come parametro) ed esegue la ricerca binaria sulla lista ordinata. Visto che binarySearch ha come input una funzione per l'ordinamento è stata implementata anche la funzione bubbleSort (che implementa ovviamente il bubble sort) che si appoggia sulla funzione sort, la quale esegue l'i-esima iterazione del bubbleSort. L'input di bubbleSort è una lista di elementi ordinabili, l'output è la stessa lista ma ordinata. L'input di sort è una lista di elementi ordinabili, l'output è la stessa lista su cui è stata eseguita un'iterazione del bubble sort. Praticamente la funzione bubbleSort non fa altro che chiamare la funzione sort tante volte quanti sono gli elementi della lista in input. Segue una lista di costrutti e caratteristiche tipici di Haskell utilizzati:

- **Pattern matching**: la funzione sort viene definita tre volte, con input diversi (lista vuota, lista con un solo elemento e lista generica), Quando sort deve essere eseguita viene scelta la prima definizione (dall'alto al basso) che corrisponde all'input fornito.
- **Funzioni condizionali**: tutte le tre funzioni sono condizionali, ovvero parte del loro comportamento dipende dal verificarsi o meno di certe condizioni. Le funzioni sort e bubbleSort utilizzano a tale scopo il costrutto **if then else** mentre la funzione binarySearch utilizza le **guards**.
- **Costruttore di liste**: la funzione sort lavora sulle liste utilizzando ":", ovvero il costrutto per costruire le liste. Infatti prende come input la lista (x:y:xs), ovvero una lista in cui x è il primo elemento, y il secondo e xs il resto della lista. Nel corpo della funzione costruisce il risultato in modo simile, per esempio x:sort(y:xs) indica una lista in cui il primo elemento è x e il restante è la lista ottenuta eseguendo sort sulla lista ottenuta concatenando y alla lista xs.
- **Ricorsione**: le tre funzioni sono ricorsive, ovvero nel corpo della funzione eseguono una chiamata alla funzione stessa. la funzione sort è "direttamente" ricorsiva, nel senso che esegue una chiamata a sort stessa, mentre binarySearch e bubbleSort contengono nel loro corpo la definizione di altre funzioni (tramite il costrutto **where**) che a loro volta sono "direttamente" ricorsive. Inoltre, la funzione search definita in binarySearch sfrutta la **tail recursion**; infatti, nel caso in cui venga effettuata una chiamata ricorsiva, questa è l'ultima cosa che viene fatta e l'output della chiamata ricorsiva è lo stesso output della funzione chiamante. Questo permette un più efficiente uso della memoria.
- **Curried Funcitions**: una funzione che ha come input una tupla con due elementi può essere trasformata in una funzione che prende un singolo elemento in input e restituisce una funzione che a sua volta prende un elemento in input e restituisce un elemento in output. Questo procedimento vale in generale per un qualsiasi numero di elementi in input. Funzioni così definite sono dette **curried**. Sono tali tutte le funzioni definite nel progetto.
- **Funzioni di ordine superiore**: sono funzioni che prendono come input altre funzioni da utilizzare nel corpo della funzione inziale. binarySearch è una funzione di ordine superiore perché prende come input una funzione di tipo [a] -> [a] (chiamata sortingAlg nel corpo di binarySearch).

Oltre a queste funzioni (che sono tutte legate tra loro) è stata implementata anche la funzione delete che accetta in input un intero e una lista di tuple del tipo (Int, Char). L'output è la lista degli interi che compaiono nella lista di tuple e che non sono uguali all'intero passato come input. Questa funzione non ha nessun senso pratico ma è stata implementata per mostrare due aspetti aggiuntivi di Haskell:

- Il risultato di delete è una lista costruita con la **list comprehension** che permette di definire liste a partire da altre liste in modo molto simile alla notazione matematica per gli insiemi.
- La lista di output è ottenuta prendendo un elemento alla volta dalla lista di input (utilizzando un **generator**), questo elemento è indicato come (i',_) ed è ovviamente una tupla. i' indica il primo elemento della tupla, che viene inserito nell'output solo se è diverso all'intero passato come input. Il secondo elemento della tupla è ininfluente alla costruzione dell'output, quindi, è indicato con il simbolo "_" che rappresenta la **wildcard**, ovvero un valore qualsiasi. Lo stesso risultato si sarebbe ottenuto costruendo la tupla (i',c'), ma sarebbe stato meno efficiente (a c' sarebbe stato assegnato un valore in memoria, mentre con l'utilizzo della wildcard questo non avviene).

Il main del programma è una serie di istruzioni, eseguite così come sarebbero eseguite in un linguaggio di programmazione imperativo. Questo è permesso dalla parola chiave **do**.

Per prima cosa nel main viene testata la funzione delete. In aggiunta, ad ogni elemento della lista restituita viene applicata (grazie alla funzione map fornita da Haskell) una **lambda funcion**, ovvero una funzione senza nome definita direttamente dove viene passata come input di un'altra funzione (in questo caso map). La lambda function definita prende un elemento x e restituisce x^x.

Successivamente vengono testate le altre funzioni implementata e l'interazione con l'utente: viene chiesta una lista di caratteri e successivamente il carattere da cercare. Dopodiché viene chiamata la funzione binarySearch con input la funzione bubbleSort ed i dati inseriti dall'utente e viene stampato su console l'output della funzione.