En este módulo vamos a especificar y programar (a partir de una prueba) el método de ORDENACIÓN DE LISTAS DE ENTEROS conocido como *isertion sort*, basado en el que viene en el libro de texto

1 Definiciones

Require Import *List.*

Require Export *ZArith.*

Open Scope Z_scope.

Primero especificamos, mediante el predicado *sorted* lo que significa que una lista esté ordenada. Se hace mediante tres axiomas (constructores) que nos garantizan tres hechos que caracterizan este significado. El primero nos dice que una lista vacía está ordenada, el segundo que una lista con un solo elemento está ordenada y, finalmente, que si una lista $z_2 :: l$ está ordenada y si $z_1 \leq z_2$, entonces la lista $z_1 :: z_2 :: l$ está también ordenada:

```
Inductive sorted : list Z → Prop :=
  | sorted0 : sorted nil
  | sorted1 : ∀ z:Z, sorted (z :: nil)
  | sorted2 :
      ∀ (z1 z2:Z) (l:list Z),
        z1 ≤ z2 →
        sorted (z2 :: l) → sorted (z1 :: z2 :: l).
```

Hint *Resolve sorted0 sorted1 sorted2 : sort.*

Después de añadir a la base de *Hints* llamada *sort* las constantes *sorted0 sorted1 sorted2* (con lo cual, cuando hagamos *auto with sort* se aplicarán también estas constantes), podemos ver algunos ejemplos. Nótese el uso de *inversion* en la prueba negativa:

```
Lemma sort_2357 :
 sorted (2 :: 3 :: 5 :: 7 :: nil).
Proof.
 auto with sort zarith.
Qed.
```

```
Lemma no_sort3425:
¬sorted (3::4::2::5::nil).
red;intro.
inversion H.
inversion H4.
```

```
auto with arith.
Qed.
```

Probemnos ahora que si una lista $z :: l$ está ordenada, entonces la lista l también lo estará:

```
Theorem sorted_inv :
 ∀ (z:Z) (l:list Z), sorted (z :: l) → sorted l.
Proof.
 intros z l H.
inversion H.
auto with sort.
assumption.
Qed.
```

2 Número de apariciones de un elemento en una lista

En esta sección programaremos primero una función que admite un elemento y una lista y nos devuelve un natural (nótese que necesitamos el *%nat* para aclarar que se trata de un *nat* porque el *Scope* abierto por defecto es *Z_scope*). En este programa utilizamos el programa de decisión *Z_eq_dec* cuyo tipo es $\forall\ x\ y : Z, \{x = y\} + \{x \neq y\}$:

```
Fixpoint nb_occ (z:Z) (l:list Z) {struct l} : nat :=
  match l with
  | nil ⇒ 0%nat
  | (z' :: l') ⇒
       match Z_eq_dec z z' with
       | left _ ⇒ S (nb_occ z l')
       | right _ ⇒ nb_occ z l'
       end
  end.
```

```
Eval compute in (nb_occ 3 (3 :: 7 :: 3 :: nil)).
```

```
Eval compute in (nb_occ 33 (3 :: 7 :: 3 :: nil)).
```

3 Permutaciones de una lista

Especifiquemos cuando un lista l' es una permutación de otra l mediante el predicado *equiv* :*list Z* → *list Z* → *Prop*:

```
Definition equiv (l l':list Z) :=
```

$\forall$ *z*:*Z*, *nb_occ z l* = *nb_occ z l'*.

Esta resulta ser una relación reflexiva simétrica y transitiva. Para probar esto último utilizaremos la constante *trans_eq* de tipo : $\forall$ (*A* : *Type*) (*x y z* : *A*), *x* = *y* $\rightarrow$ *y* = *z* $\rightarrow$ *x* = *z* que nos garantiza la transitividad de la igualdad.

Lemma *equiv_refl* : $\forall$ *l*:*list Z*, *equiv l l*.
Proof.
 unfold equiv; *trivial*.
Qed.

Lemma *equiv_sym* : $\forall$ *l l'*:*list Z*, *equiv l l'* $\rightarrow$ *equiv l' l*.
Proof.
  unfold equiv; *auto*.
Qed.

Lemma *equiv_trans* :
 $\forall$ *l l' l"*:*list Z*, *equiv l l'* $\rightarrow$
   equiv l' l" $\rightarrow$
   equiv l l".
Proof.
 intros l l' l" H H0 z.
 eapply trans_eq; *eauto*.
Qed.

Se cumple también la congruencia de la equivalencia con el constructor *cons*:

Lemma *equiv_cons* :
 $\forall$ (*z*:*Z*) (*l l'*:*list Z*), *equiv l l'* $\rightarrow$
   equiv (*z* :: *l*) (*z* :: *l'*).
Proof.
 intros z l l' H z'.
 simpl; *case* (*Z_eq_dec z' z*); *auto*.
Qed.

y la siguiente propiedad:

Lemma *equiv_perm* :
 $\forall$ (*a b*:*Z*) (*l l'*:*list Z*),
   equiv l l' $\rightarrow$
   equiv (*a* :: *b* :: *l*) (*b* :: *a* :: *l'*).
Proof.
 intros a b l l' H z; *simpl*.
 case (*Z_eq_dec z a*); *case* (*Z_eq_dec z b*);
  simpl; *case* (*H z*); *auto*.
Qed.

Hint *Resolve equiv_cons equiv_refl equiv_perm* : *sort*.

```
Fixpoint insertion (z:Z) (l:list Z) {struct l} : list Z :=
  match l with
  | nil ⇒ z :: nil
  | cons a l' ⇒
      match Z_le_gt_dec z a with
      | left _ ⇒ z :: a :: l'
      | right _ ⇒ a :: (insertion z l')
      end
  end.
```

```
Eval compute in (insertion 4 (2 :: 5 :: nil)).
```

```
Eval compute in (insertion 4 (24 :: 50 ::nil)).
```

Esta función tiene las siguientes propiedades que confirman que es adecuada para ordenar listas de acuerdo con nuestra especificación

```
Lemma insertion_equiv : ∀ (l:list Z) (x:Z),
                        equiv (x :: l) (insertion x l).
Proof.
 induction l as [|a l0 H]; simpl ; auto with sort.
 intros x; case (Z_le_gt_dec x a);
   simpl; auto with sort.
 intro; apply equiv_trans with (a :: x :: l0);
   auto with sort.
Qed.
```

```
Lemma insertion_sorted :
 ∀ (l:list Z) (x:Z), sorted l → sorted (insertion x l).
Proof.
 intros l x H; elim H; simpl; auto with sort.
 intro z; case (Z_le_gt_dec x z); simpl;
   auto with sort zarith.
 intros z1 z2; case (Z_le_gt_dec x z2); simpl; intros;
  case (Z_le_gt_dec x z1); simpl; auto with sort zarith.
Qed.
```

Finalmente, definimos la función que ordena listas

```
Definition sort :
  ∀ l:list Z, {l' : list Z | equiv l l' ∧ sorted l'}.
 induction l as [| a l IHl].
 ∃ (nil (A:=Z)); split; auto with sort.
 case IHl; intros l' [H0 H1].
 ∃ (insertion a l'); split.
 apply equiv_trans with (a :: l'); auto with sort.
 apply insertion_equiv.
```

apply insertion_sorted; *auto.*
Defined.

Definition *pr1* $(A : Set)$ $(P : A \rightarrow Prop)$ $(H : \{x : A \mid P\ x\})$:=
let (a, p) := *H in a.*
Implicit *Arguments pr1* [A].

Definition *pr2* $(A : Set)$ $(P : A \rightarrow Prop)$ $(H : \{x : A \mid P\ x\})$:=
let (a, p) *as H return* (*P* (*pr1 P H*)) := *H in p.*

Implicit *Arguments pr2* [A].

Definition *Sort*:=*fun* (*l*:*list Z*) ⇒ *pr1* (*fun* (*l'*:*list Z*)=>
equiv l l' ∧ *sorted l'*) (*sort l*).

Eval compute in sort (3::1::4::*nil*).

Eval compute in Sort (3::6::21::-4::0::4::*nil*).

4 Ejercicios

Inductive *lelist* (*a*:*Z*) : *list Z* → *Prop* :=
| *nil_le* : *lelist a nil*
| *cons_le* : ∀ (*b*:*Z*) (*l*:*list Z*), $a \leq b$ → *lelist a* (*b* :: *l*).

Hint *Constructors lelist.*

Definition *lista1*:=3::1::7::8::*nil.*
Lemma *k*: (*lelist* 1 *lista1*).
Check cons_le.
unfold lista1.
apply (*cons_le* 1 3).
auto with zarith.
Qed.

Lemma *minv*:∀ (*a*:*Z*) (*l*: *list Z*), (*lelist a l*)->(*l*=*nil*) ∨ (∃ *b*:*Z*,∃ *l'*:*list Z*,
l=*b*::*l'*/\($a \leq b$)).
Admitted.

Hacer la siguiente prueba utilizando el lema de inversion anterior *minv* y sin usar *inversion*:
Lemma *lelist_inv'* : ∀ (*a b*:*Z*) (*l*:*list Z*), *lelist a* (*b* :: *l*) → $a \leq b$.
Proof.

Admitted.

Hacerlo usando inversión de H. Le cambio el nombre para que no coincida con el anterior:

Lemma $lelist\_inv$: $\forall$ $(a\ b{:}Z)$ $(l{:}list\ Z)$, $lelist\ a\ (b :: l) \rightarrow a \le b$.
Proof.
Admitted.

Lemma $sorted\_lelist$: $\forall$ $(a{:}Z)$ $(l{:}list\ Z)$, $sorted\ l \rightarrow lelist\ a\ l \rightarrow$
$sorted\ (a{::}l)$.
Proof.
Admitted.

Una nueva definición de sorted

Inductive ord : $list\ Z \rightarrow Prop$:=
  | nil_sort : $ord\ nil$
  | $cons\_sort$:
    $\forall$ $(a{:}Z)$ $(l{:}list\ Z)$, $ord\ l \rightarrow lelist\ a\ l \rightarrow ord\ (a :: l)$.
Hint *Constructors ord.*

Lemma ord_inv :
 $\forall$ $(a{:}Z)$ $(l{:}list\ Z)$, $ord\ (a :: l) \rightarrow ord\ l \wedge lelist\ a\ l$.
Proof.
intros; *inversion* H; *auto with datatypes.*
Qed.

Lemma ord_singl:$\forall$ $(z{:}Z)$, $ord\ (z{::}nil)$.
auto with sort datatypes.
Qed.

Lemma ord_rec :
 $\forall$ $P{:}list\ Z \rightarrow Set$,
   $P\ nil \rightarrow$
   $(\forall\ (a{:}Z)\ (l{:}list\ Z),\ ord\ l \rightarrow P\ l \rightarrow lelist\ a\ l \rightarrow P\ (a :: l)) \rightarrow$
   $\forall\ y{:}list\ Z,\ ord\ y \rightarrow P\ y$.
Proof.
Admitted.

Probar que *ord* y *sorted* son equivalentes

Lemma ord_sort: $\forall$ $l{:}list\ Z$, $ord\ l \rightarrow sorted\ l$.
Proof.
Admitted.

Lemma $sort\_ord$:$\forall$ $l{:}list\ Z$, $sorted\ l\rightarrow ord\ l$.
Proof.
Admitted.