1.

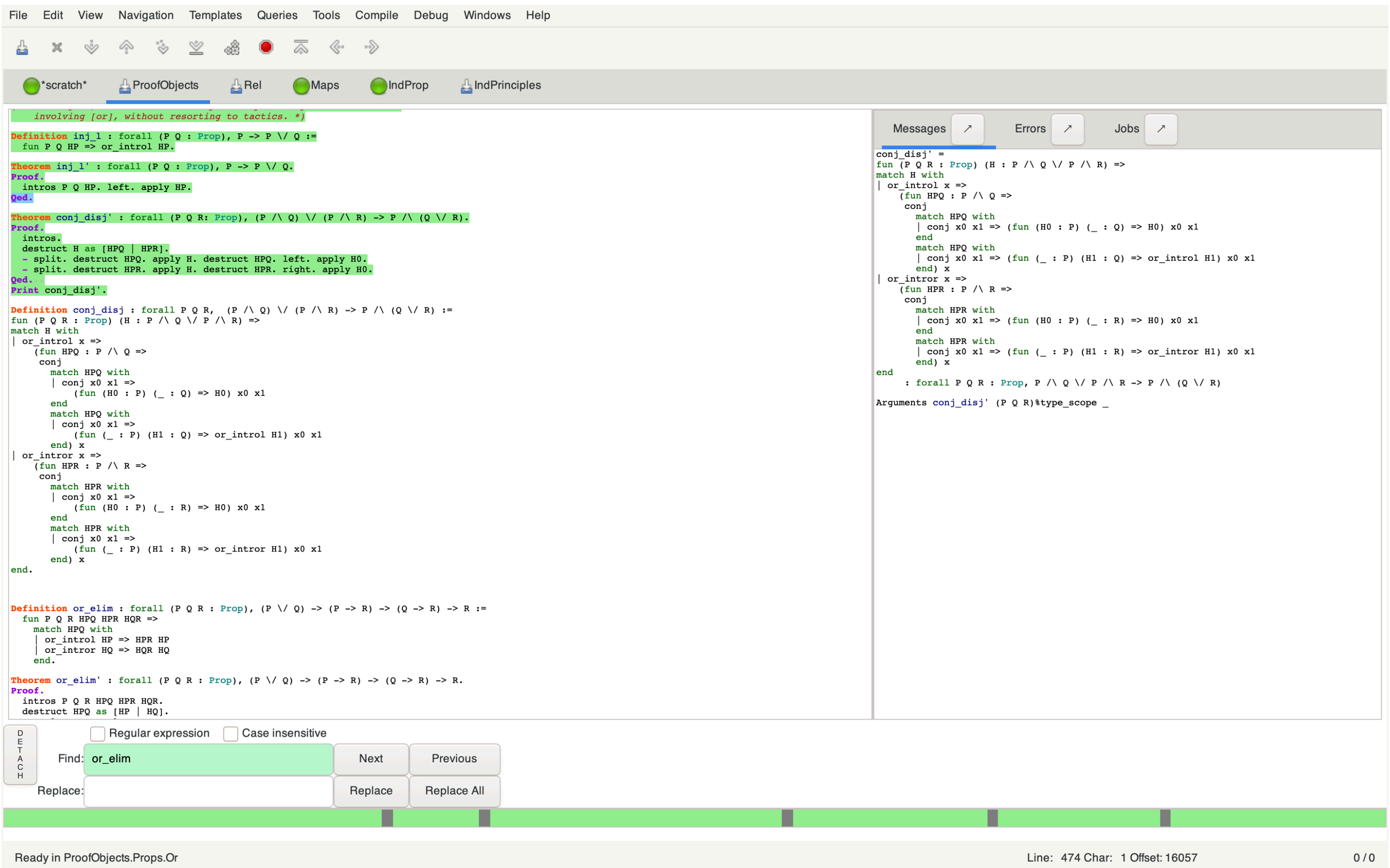
```
Definition inj_l : forall (P Q : Prop), P -> P \/ Q :=
  fun P Q HP => or_introl HP.

Theorem inj_l' : forall (P Q : Prop), P -> P \/ Q.
Proof.
  intros P Q HP. left. apply HP.
Qed.

Theorem conj_disj' : forall (P Q R: Prop), (P /\ Q) \/ (P /\ R) -> P /\ (Q \/ R).
Proof.
  intros.
  destruct H as [HPQ | HPR].
  - split. destruct HPQ. apply H. destruct HPQ. left. apply H0.
  - split. destruct HPR. apply H. destruct HPR. right. apply H0.
Qed.
Print conj_disj'.

Definition conj_disj : forall P Q R,  (P /\ Q) \/ (P /\ R) -> P /\ (Q \/ R) :=
fun (P Q R : Prop) (H : P /\ Q \/ P /\ R) =>
match H with
| or_introl x =>
    (fun HPQ : P /\ Q =>
     conj
       match HPQ with
       | conj x0 x1 =>
           (fun (H0 : P) (_ : Q) => H0) x0 x1
       end
       match HPQ with
       | conj x0 x1 =>
           (fun (_ : P) (H1 : Q) => or_introl H1) x0 x1
       end) x
| or_intror x =>
    (fun HPR : P /\ R =>
     conj
```

```
        match HPR with
        | conj x0 x1 =>
            (fun (H0 : P) (_ : R) => H0) x0 x1
        end
        match HPR with
        | conj x0 x1 =>
            (fun (_ : P) (H1 : R) => or_intror H1) x0 x1
        end) x
end.
```



```
Definition conj_disj : forall P Q R,  (P /\ Q) \/ (P /\ R) -> P /\ (Q \/ R) :=
fun (P Q R : Prop) (H : P /\ Q \/ P /\ R) =>
match H with
| or_introl x =>
    (fun HPQ : P /\ Q =>
     conj
       match HPQ with
       | conj x0 x1 =>
           (fun (H0 : P) (_ : Q) => H0) x0 x1
       end
       match HPQ with
       | conj x0 x1 =>
           (fun (_ : P) (H1 : Q) => or_introl H1) x0 x1
       end) x
| or_intror x =>
    (fun HPR : P /\ R =>
     conj
       match HPR with
       | conj x0 x1 =>
           (fun (H0 : P) (_ : R) => H0) x0 x1
       end
       match HPR with
       | conj x0 x1 =>
           (fun (_ : P) (H1 : R) => or_intror H1) x0 x1
       end) x
```

2.

```
Theorem plus_one_r' : forall n:nat,
  n + 1 = S n.
Proof.
  apply nat_ind.
  - reflexivity.
  - simpl. intros n IH.
    rewrite IH. reflexivity.
Qed.
```

```
Theorem plus_one_r' : forall n:nat,
  n + 1 = S n.
Proof.
  apply nat_ind.
  - reflexivity.
  - simpl. intros n IH.
    rewrite IH. reflexivity.
Qed.
```

3.

```
Theorem lt_trans' :
  transitive lt.
Proof.
  (* Prove this by induction on evidence that [m] is less than [o]. *)
  unfold lt. unfold transitive.
  intros n m o Hnm Hmo.
  induction Hmo as [| m' Hm'o].
  - apply le_S. apply Hnm.
  - apply le_S. apply IHHm'o.
Qed.
```

```
(** **** Exercise: 2 stars, standard, optional (le_trans_hard_way)

    We can also prove [lt_trans] more laboriously by induction,
    without using [le_trans].  Do this. *)

Theorem lt_trans' :
  transitive lt.
Proof.
  (* Prove this by induction on evidence that [m] is less than [o]. *)
  unfold lt. unfold transitive.
  intros n m o Hnm Hmo.
  induction Hmo as [| m' Hm'o].
  - apply le_S. apply Hnm.
  - apply le_S. apply IHHm'o.
Qed.
```

4.

```
Lemma rsc_trans :
  forall (X:Type) (R: relation X) (x y z : X),
      clos_refl_trans_1n R x y  ->
      clos_refl_trans_1n R y z ->
      clos_refl_trans_1n R x z.
Proof.
  intros X R x y z xy.
  generalize dependent z.
  induction xy as [| x y' y H H' IH].
  - trivial.
  - intros z yz.
    apply (rt1n_trans R x y' z).
    + assumption.
```

```
    + apply IH. assumption.
Qed.
```

```
Lemma rsc_trans :
  forall (X:Type) (R: relation X) (x y z : X),
      clos_refl_trans_1n R x y  ->
      clos_refl_trans_1n R y z ->
      clos_refl_trans_1n R x z.
Proof.
  intros X R x y z xy.
  generalize dependent z.
  induction xy as [| x y' y H H' IH].
  - trivial.
  - intros z yz.
    apply (rt1n_trans R x y' z).
    + assumption.
    + apply IH. assumption.
Qed.
```