let rec addLen n = function
| [] -> n
| x :: xs -> addLen (n+1) xs

Example:
addLen 0 [1,2,3]

Calling addLen with same arguments will always produce the same result. We can infer result through function expansion and reduction of expressions. This allows us to:

→ Prove algorithm correctness
→ Understand and predict algorithm outcome
Procedural Programming

Procedural programs can change the machine state.
They can interact with its environment.
They use control structures like branching, iteration and procedures.
They use data abstractions of the computer’s memory:
- references to memory cells
- arrays: blocks of memory cells
- linked structures, especially linked lists

a) update variable / array
b) sending / receiving data

concept: memory cells that are mutable
What are References?

In functional programming:
The store is an *invisible* device inside the computer.

In procedural / imperative programming:
The store is *visible*.
What are References?

In functional programming:
The store is an **invisible** device inside the computer

In procedural / imperative programming:
The store is **visible**

- References are *storage locations*
- They can be:
  (a) created
  (b) inspected
  (c) updated

The box has an address
**ML Primitives for References**

\[ \tau \text{ ref} \quad \text{type of references to type } \tau \]

\[ \text{ref } E \quad \text{create a reference} \]

\[ \text{initial contents} = \text{the value of } E \]

\[ ! P \quad \text{return the current contents of reference } P \quad \text{‘dereferencing’} \]

\[ P := E \quad \text{update the contents of } P \text{ to the value of } E \]
ML Primitives for References

\[ \tau \text{ ref} \quad \text{type of references to type } \tau \]

\[ \text{ref } E \quad \text{create a reference} \]
\[ \text{initial contents} = \text{the value of } E \]

\[ ! P \quad \text{return the current contents of reference } P \quad \text{\textquote{derefencing}} \]

\[ P := E \quad \text{update the contents of } P \text{ to the value of } E \]

\[ \text{P for \textquote{pointer}} \]

\[ \text{pointer to a \textquote{box}} \]

\[ \text{contents of that \textquote{box}} \]
ML Primitives for References

\[ \tau \text{ ref} \] type of references to type \( \tau \)

\[ \text{ref } E \] create a reference

initial contents = the value of \( E \)

\[ ! P \] return the current contents of reference \( P \) 'dereferencing'

\[ P := E \] update the contents of \( P \) to the value of \( E \)

Three new ML functions / operators:

\[ \text{ref} : \ 'a \rightarrow \ 'a \text{ ref} \] (a) create box

\[ ! : \ 'a \text{ ref} \rightarrow \ 'a \] (b) inspect box content

\[ := : \ 'a \text{ ref} \rightarrow \ 'a \rightarrow \text{unit} \] (c) update box content
# let p = ref 5 (* create a reference *)
val p : int ref = {contents = 5}

# p := !p + 1 (* p now holds value 6 *)
- : unit = ()

# let ps = [ ref 77; p ]
val ps : int ref list = [{contents = 77}; {contents = 6}]

# List.hd ps := 3
- : unit = ()

# ps
- : int ref list = [{contents = 3}; {contents = 6}]
Aliasing: two values refer to the same mutable cell

```
# let p = ref 5 (* create a reference *)
val p : int ref = {contents = 5}

# let z = p
val z : int ref = {contents = 5}

# p := !p + 1  (* p now holds value 6 *)
- : unit = ()

# p
- : int ref = {contents = 6}

# z
- : int ref = {contents = 6}
```
Commands: Expressions with Effects

• Basic commands update references, write to files, etc.

• $C_1; \ldots; C_n$ causes a series of expressions to be evaluated and returns the value of $C_n$.

• A typical command returns the empty tuple: ()

• if $B$ then $C_1$ else $C_2$ behaves like the traditional control structure if $C_1$ and $C_2$ have effects.

• Other ML constructs behave naturally with commands, including match expressions and recursive functions.
Commands: Expressions with Effects

- Basic commands update references, write to files, etc.
- $C_1; \ldots; C_n$ causes a series of expressions to be evaluated and returns the value of $C_n$.
- A typical command returns the empty tuple: ()
- `if B then C_1 else C_2` behaves like the traditional control structure if $C_1$ and $C_2$ have effects.
- Other ML constructs behave naturally with commands, including `match` expressions and recursive functions.

Example:

```ml
> 1 + (print_endline "abc"; 3; 101);

abc
- : int = 102
```
Example: length without Mutability

```
let rec addLen n = function
    | []    -> n
    | x :: xs -> addLen (n+1) xs
```

```
addLen 0 [1,2,3]
addLen 1 [2,3]
addLen 2 [3]
addLen 3 []
==> returns 3
```
Iteration: the while Command

```ocaml
# let tlopt = function
    | [] -> None
    | _::xs -> Some xs
val tlopt : 'a list -> 'a list option = <fun>

# let length xs =
    let lp = ref xs in (* list of uncounted elements *)
    let np = ref 0 in (* accumulated count *)
    let fin = ref false in
    while not !fin do
        match tlopt !lp with
        | None -> fin := true
        | Some xs ->
            lp := xs;
            np := 1 + !np
        done;

    !np (* the final count is returned *)
val length : 'a list -> int = <fun>
```
# let tlopt = function
| []    -> None
| _::xs -> Some xs
val tlopt : 'a list -> 'a list option = <fun>

# let length xs =
    let lp  = ref xs in (* list of uncounted elements *)
    let np  = ref 0  in (* accumulated count *)
    let fin = ref false in
    while not !fin do
        match tlopt !lp with
        | None -> fin := true
        | Some xs ->
            lp := xs;
            np := 1 + !np
        done;
    !np (* the final count is returned *)
val length : 'a list -> int = <fun>
# let tlopt = function
| []   -> None
| _::xs -> Some xs
val tlopt : 'a list -> 'a list option = <fun>

# let length xs =
  let lp = ref xs in (* list of uncounted elements *)
  let np = ref 0 in (* accumulated count *)
  let fin = ref false in
  while not !fin do
    match tlopt !lp with
    | None -> fin := true
    | Some xs ->
      lp := xs;
      np := 1 + !np
    done;
    !np (* the final count is returned *)
val length : 'a list -> int = <fun>
Iteration: the while Command

# let tlopt = function
|  []  -> None
| _::xs -> Some xs
val tlopt : 'a list -> 'a list option = <fun>

# let length xs =
  let lp  = ref xs in (* list of uncounted elements *)
  let np  = ref 0  in (* accumulated count *)
  let fin = ref false in
  while not !fin do
    match tlopt !lp with
    | None -> fin := true
    | Some xs ->
      lp := xs;
      np := 1 + !np
  done;
  !np (* the final count is returned *)
val length : 'a list -> int = <fun>
Iteration: the while Command

# let tlopt = function
    | [] -> None
    | _::xs -> Some xs
val tlopt : 'a list -> 'a list option = <fun>

# let length xs =
    let lp = ref xs in (* list of uncounted elements *)
    let np = ref 0 in (* accumulated count *)
    let fin = ref false in
    while not !fin do
        match tlopt !lp with
            | None -> fin := true
            | Some xs ->
                lp := xs;
                np := 1 + !np
        done;
    !np (* the final count is returned *)
val length : 'a list -> int = <fun>
Example: length with Mutability

evaluation steps:

\[
\text{length}([1;2;3]) \\
\Rightarrow \text{lp} = \text{ref [1,2,3]}
\]

\[
\text{tlopt [1;2;3] \neq None} \Rightarrow \text{true} \\
\text{lp := [2,3]; np := 1+0;}
\]

\[
\text{tlopt [2;3] \neq None} \Rightarrow \text{true} \\
\text{lp := [3]; np := 1+1}
\]

\[
\text{tlopt [3] \neq None} \Rightarrow \text{true} \\
\text{lp := []; np := 1+2}
\]

\[
\text{tlopt [ ] \neq None} \Rightarrow \text{false} \\
\text{fin := true} \\
\Rightarrow \text{return !np} \\
\Rightarrow \text{returns 3}
\]
let tlopt = function
    | []  -> None
    | _::xs -> Some xs

let length xs =
    let lp  = ref xs in
    let np  = ref 0  in
    let fin = ref false in
    while not !fin do
        match tlopt !lp with
        | None -> fin := true
        | Some xs ->
            lp := xs;
            np := 1 + !np
    done;
    !np
val length : 'a list -> int = <fun>

let rec addLen n =
    function
        | []  -> n
        | x :: xs ->
            addLen (n+1) xs
# exception TooMuch of int

exception TooMuch of int

# let makeAccount initBalance =

let balance = ref initBalance in

let withdraw amt =
  if amt > !balance then
    raise (TooMuch (amt - !balance))
  else begin
    balance := !balance - amt;
    !balance
  end

in

withdraw

val makeAccount : int -> int -> int = <fun>
# exception TooMuch of int

exception TooMuch of int

# let makeAccount initBalance =
let balance = ref initBalance in
let withdraw amt =
  if amt > !balance then
    raise (TooMuch (amt - !balance))
  else begin
    balance := !balance - amt;
    !balance
  end
in
withdraw
val makeAccount : int -> int -> int = <fun>

returns a function that returns contents of ‘balance’, not the cell itself
# exception TooMuch of int
exception TooMuch of int
# let makeAccount initBalance =
   let balance = ref initBalance in
   let withdraw amt =
     if amt > !balance then
       raise (TooMuch (amt - !balance))
     else begin
         balance := !balance - amt;
         !balance
     end
   in
   withdraw
val makeAccount : int -> int -> int = <fun>

balance never escapes the definition of makeAccount
returns a function that returns contents of 'balance', not the cell itself
let my_account = makeAccount 30;
my_account : int -> int = <fun>

let my_new_balance = my_account 10;
my_new_balance : int = 20

let my_new_balance = my_account ~10;
my_new_balance : int = 30
Two Bank Accounts

# let student = makeAccount 500
val student : int -> int = <fun>

# let director = makeAccount 4000000
val director : int -> int = <fun>

# student 5 (* coach fare *)
- : int = 495

# director 150000 (* Tesla *)
- : int = 3850000

# student 500 (* oh oh *)
Exception: TooMuch 5.
# 

```ml
# ["a"; "b"; "c"]
(* allocate a fresh string array *)
- : string array = ["a"; "b"; "c"]

# Array.make 3 'a'
(* array of size 3 with cell containing 'a' *)
- : char array = ['a'; 'a'; 'a']

# let aa = Array.init 5 (fun i -> i * 10)
(* array of size 5 initialised to (fun i) *)
val aa : int array = [0; 10; 20; 30; 40]

# Array.get aa 3
(* retrieve the 4th cell in the array *)
- : int = 30

# Array.set aa 3 42
(* set the 4th cell's value to 42 *)
- : unit = ()
```
Array Examples

# Array.make
- : int -> 'a -> 'a array = <fun>

# Array.init
- : int -> (int -> 'a) -> 'a array = <fun>

# Array.get
- : 'a array -> int -> 'a = <fun>

# Array.set
- : 'a array -> int -> 'a -> unit = <fun>
We must write !p to get the contents of p
We write just p for the address of p

We can store private reference cells in functions; simulating object oriented programming

OCaml's assignment syntax is
V := E instead of V = E

OCaml has similar control structures: while/done, for/done and match/with

OCaml has short syntax for updating arrays x.(1) and the access is safe against buffer overflows
What More Is There to ML?

With references, we can now make mutable linked lists

```ocaml
# type 'a mlist =
| Nil
| Cons of 'a * 'a mlist ref
```

```
type 'a mlist = Nil | Cons of 'a * 'a mlist ref
```

![Linked list diagram](image)
References to References

Two ways to visualize references to references:

(1) Using pointers:

(2) Using nested boxes:
Linked (Mutable) Lists

→ The tail can be redirected!

# type 'a mlist =
  | Nil
  | Cons of 'a * 'a mlist ref

type 'a mlist = Nil | Cons of 'a * 'a mlist ref

# let rec mlistOf = function
  | [] -> Nil
  | x :: l -> Cons (x, ref (mlistOf l))

mlist : 'a list -> 'a mlist = <fun>

creates a new pointer to rest of mlist
Extending a List to the Rear

pointing to a 'box'

```ocaml
# let extend mlp x =
    let last = ref Nil in
    mlp := Cons (x, last);
    last
> val extend = fn : 'a mlist ref * 'a -> 'a mlist ref
```
Example of Extending a List

```ocaml
# let mlp = ref (Nil: string mlist);;
val mlp : string mlist ref = {contents = Nil}

# extend mlp "a";;
- : string mlist ref = {contents = Nil}

# let mlp = ref (Nil : string mlist);;
val mlp : string mlist ref = {contents = Nil}

# let it = extend mlp "a" ;;
val it : string mlist ref = {contents = Nil}

# extend it "b" ;;
- : string mlist ref = {contents = Nil}

# mlp ;;
- : string mlist ref =
{contents = Cons ("a",
    {contents = Cons ("b", {contents = Nil}))})
```
ref (Cons (x, ref (Cons (y, ref Nil)))))
Destructive Concatenation

pointing to a 'box'

contents of a 'box'

# let rec joining mlp ml2 =
  match !mlp with
  | Nil -> mlp := ml2
  | Cons (_, mlp1) -> joining mlp1 ml2
val joining : 'a mlist ref * 'a mlist -> unit = <fun>

# let join ml1 ml2 =
  let mlp = ref ml1 in
  joining mlp ml2;
  !mlp
val join : 'a mlist -> 'a mlist -> 'a mlist = <fun>
What does this return?

- : string mlist =
  Cons ("a",
    {contents = Cons ("b",
      {contents = Cons ("c", {contents = Nil})})})