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 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
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
The slide presents the ML primitives, but most languages have analogues so of them, often heavily disguised. We need a means of creating references (or allocating storage), getting at the current contents of a reference cell, and updating that cell.

The function `ref` creates references (also called locations). Calling `ref` allocates a new location in memory. Initially, this location holds the value given by expression `E`. Although `ref` is an ML function, it is not a function in the mathematical sense. For example, `ref(0)=ref(0)` evaluates to false.

The function `!`, when applied to a reference, returns its contents. This operation is called dereferencing. Clearly `!` is not a mathematical function; its result depends upon the store.

The assignment `P := E` evaluates expression `P`, which must be a reference `P`, and `E`. It stores at address `P` the value of `E`. Syntactically, `:=` is a function and `P := E` is an expression, even though it updates the store. Like many functions that change the state, it returns the value () of type `unit`.

If `τ` is some ML type, then `τ ref` is the type of references to cells that can hold values of `τ`. Please don’t confuse the type `ref` with the function `ref`. This table of the primitive functions and their types might be useful:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ref</code></td>
<td>create a reference</td>
</tr>
<tr>
<td><code>E</code></td>
<td><code>ref E</code></td>
</tr>
<tr>
<td><code>P</code></td>
<td>initial contents = the value of <code>E</code></td>
</tr>
<tr>
<td><code>! P</code></td>
<td>return the current contents of reference <code>P</code></td>
</tr>
<tr>
<td><code>P := E</code></td>
<td>update the contents of <code>P</code> to the value of <code>E</code></td>
</tr>
</tbody>
</table>
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 -> 'a ref \hspace{1cm} \text{(a) create box}

! : 'a ref -> 'a \hspace{1cm} \text{(b) inspect box content}

:= : 'a ref -> 'a -> unit \hspace{1cm} \text{(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: ()

• $\text{if } B \text{ then } C_1 \text{ else } C_2$ 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: $> 1 + (\text{print_endline } "abc"; 3; 101);$
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: ()
- \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ behaves like the traditional control structure if } C_1 \text{ and } C_2 \text{ have effects.}
- Other ML constructs behave naturally with commands, including \texttt{match} expressions and recursive functions.

Example:

$> 1 + (\text{print}\_\text{endline} \ "abc"; 3; 101);$

```
abc
abc
- : \texttt{int} = 102
```
Example: `length` without Mutability

```ocaml
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

# 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

```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>
```
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
  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

```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>
```
Example: \texttt{length with Mutability}

evaluation steps:

\texttt{length([1;2;3])}

\texttt{===> \texttt{lp = ref [1,2,3]}}
Example: \texttt{length with Mutability}

evaluation steps:

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

\texttt{tlopt [1;2;3] != None} \implies \texttt{true}
\texttt{lp := [2,3]; np := 1+0;}
Example: `length` with Mutability

evaluation steps:

```
length([1;2;3])
==> lp = ref [1,2,3]

tlopt [1;2;3] != None ==> true
lp := [2,3]; np := 1+0;

tlopt [2;3] != None ==> true
lp := [3]; np := 1+1
```
Example: length with Mutability

evaluation steps:

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

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

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

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

evaluation steps:

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

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

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

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

\[
\text{tlopt} \ [.] \neq \text{None} \quad \Rightarrow \quad \text{false}
\]
\[
\text{fin} := \text{true}
\]
\[
===> \text{return} \ !\ np
\]
\[
===> \text{returns} \ 3
\]
# 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 - amd;
      !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>

returns a function that returns contents of 'balance', not the cell itself

balance never escapes the definition of makeAccount
let my_account = makeAccount 30;
let my_account = makeAccount 30;

my_account : int -> int = <fun>
let my_account = makeAccount 30;

my_account : int -> int = <fun>

let my_new_balance = my_account 10;
let my_account = makeAccount 30;
my_account : int -> int = <fun>

let my_new_balance = my_account 10;
my_new_balance : int = 20
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;
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
# 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 Primitives for Arrays

# ["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 \( \mathsf{!p} \) 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
```

```
3 5 9 Nil
```

```
7
```
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

→ The tail can be redirected!

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

mlist : 'a list -> 'a mlist = <fun>
Extending a List to the Rear

pointing to a ‘box’

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

# 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}))})}
\[ mlp \arrow{x} \arrow{\text{Nil}} \\
\text{Nil} \arrow{\text{Nil}} \arrow{\text{Nil}} \\
\]
ref (Cons (x, ref (Cons (y, ref Nil)))))
Destructive Concatenation

```ocaml
# 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>
```
## Side-Effects

```ocaml
# let ml1 = mlistOf ["a"];;
val ml1 : string mlist = Cons ("a", {contents = Nil})
# let ml2 = mlistOf ["b";"c"];;
val ml2 : string mlist =
  Cons ("b", {contents = Cons ("c", {contents = Nil})})
# join ml1 ml2 ;;
```

What does this return?
What does this return?

```
# let ml1 = mlistOf ['a'];;
val ml1 : string mlist = Cons ('a', {contents = Nil})
# let ml2 = mlistOf ['b';'c'];;
val ml2 : string mlist =
    Cons ('b', {contents = Cons ('c', {contents = Nil})})
# join ml1 ml2 ;;
- : string mlist =
  Cons ('a',
    {contents = Cons ('b',
                      {contents = Cons ('c', {contents = Nil}))})})
```