

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

<http://www.cs.uiuc.edu/class/fa06/cs421/>

Based in part on slides by Mattox Beckman, as updated
by Vikram Adve and Gul Agha

User Defined Types

- Records and variants allow for more accurate modeling of needed data structures than can be had from just arrays and pairs
- Better modeling leads to better encapsulation

Data Structures in a Functional Language

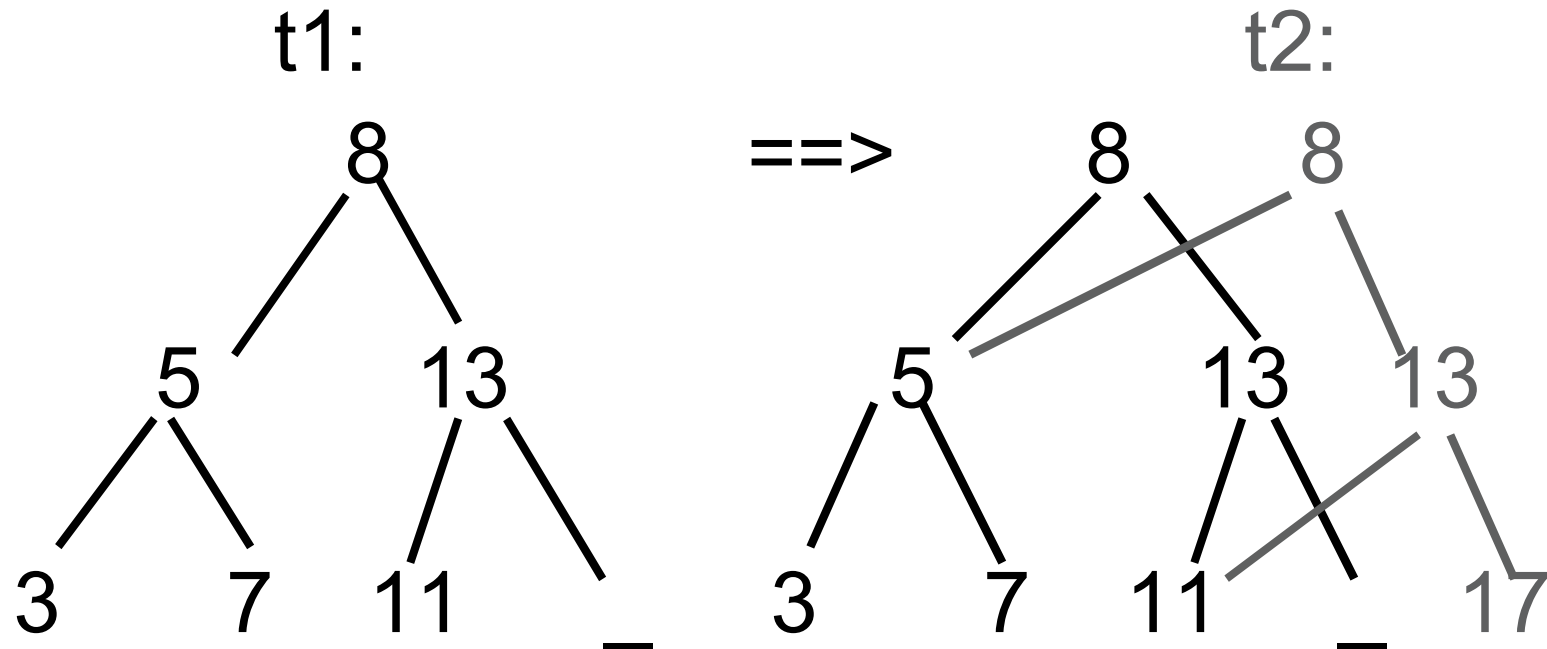
```
type tree = Node of int * tree * tree | Leaf  
of int | Empty
```

```
let t1 =
```

```
    Node(8, Node(5, Leaf 3, Leaf 7),  
          Node(13, Leaf 11, Empty))
```

```
let t2 = add t1 17
```

Data Structures in a Functional Language



- Empty --> Leaf 17
- Must rebuild Node 13 and Node 8

Data Structures with Pointers

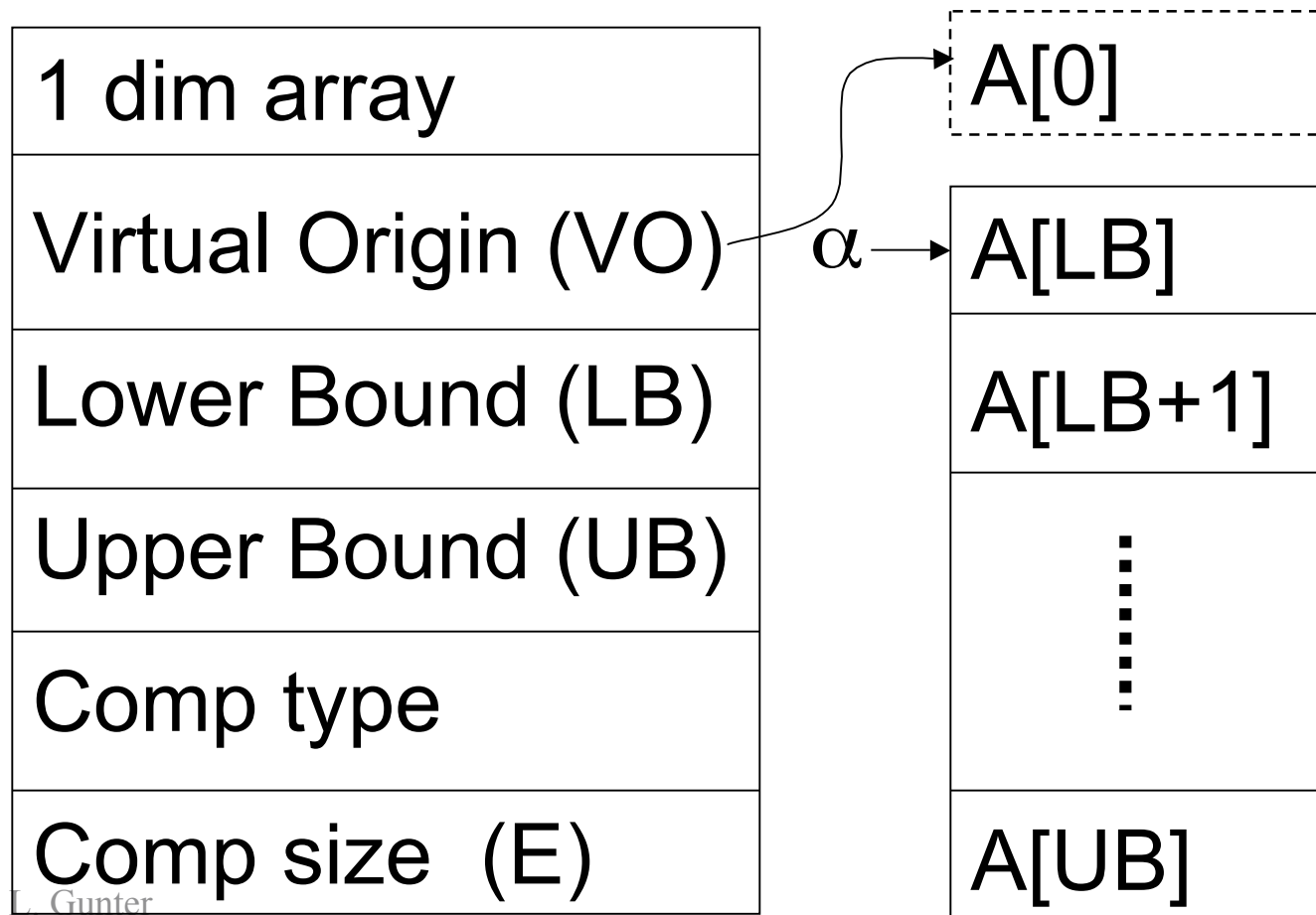
```
type mtree={n: int; mutable left: mtree option;  
  mutable right: mtree option};;  
• let t1= {n=8;  
  left= Some {n=5;left= Some {n=3; left=None;  
                                right=None};  
    right=Some {n=7; left=None;  
                right=None}};  
  right=Some {n=13; left= Some {n=11; left=None;  
                                right=None};  
    right=None}};;
```


Arrays

- Ordered sequence of fixed number of objects all of the same type
- Indexed by integer, subrange, or enumeration type, called subscript
- Multidimensional arrays have one subscript per each dimension
- L-value for array element given by accessing formula

Array Layout

- Assume one dimension



Array Component Access

- Component access through subscripting, both for lookup (r-value) and for update (l-value)
- Component access should take constant time (ie. looking up the 5th element takes same time as looking up 100th element)

Summary: Records, Variant Records, Pointers, Arrays

- Record types are good for grouping heterogeneous data and naming them
 - All instances have identical structure
 - Easy to access field information - constant time
- Disjoint types (variants) are good for types that have multiple structures.
 - Allows for more abstract treatment of data
 - Allows for recursive user-defined types

Summary: Records, Variant Records, Pointers, Arrays

- Pointer types allow explicit management of memory
 - Allows 'in-place' modifications to large data structures
 - Limits structure sharing
- Array types are good for many homogenous data values.
 - Allows efficient memory organization
 - Allows efficient memory access
 - Allows proving independent accesses:
 $A[i][j]$, $A[i+1][j]$

Information Hiding - Encapsulation

- Consider the C code:

```
typedef struct RationalType {  
    int numerator;  
    int denominator;  
} Rational  
Rational mk_rat (int n,int d) { ...}  
Rational add_rat (Rational x, Rational y)  
{ ... }
```
- Can use `mk_rat`, `add_rat` without knowing the details of `RationalType`

Need for Abstract Types

- Problem with previous example: abstraction not enforced
 - User can create Rationals without using `mk_rat`
 - User can access and alter numerator and denominator directly without using provided functions

SML -> Ocaml

- SML (still) supports a mechanism for creating abstract types without needing the module system
- Modules can implement abstract types, so Ocaml doesn't support simple abstract type any more
- fun ----> let rec
- val ----> let
- case _ of _ => _ ----> match _ with _->_

Abstract Types in SML

```
abstype rational =
```

```
  C of {numerator:int, denominator:int} with
```

```
  fun mk_rat (n,d) = case d of 0 => raise Div
```

```
    | _ => C {numerator = n,  
             denominator = d}
```

```
  fun add_rat
```

```
    (C{numerator=n1,denominator=d1},
```

```
     C{numerator=n2,denominator=d2}) =
```

```
  C{numerator = n1 * d2 + n2 * d1,
```

```
     denominator = d1 * d2}
```

```
end;
```

Abstract Types in SML

```
type rational
```

```
val mk_rat = fn : int * int -> rational
```

```
val add_rat = fn : rational * rational ->  
  rational
```

```
val half = mk_rat(1,2);
```

```
val half = - : rational
```

```
case half of C info => info;
```

```
stdIn:4.1-4.28 Error: non-constructor  
  applied to argument in pattern: C
```

Abstract Types in SML

- Can create objects of type rational
- Can use associated functions defined in abstype
- Can **not** use the implementation
 - Constructor C not in scope - only in scope inside interface function definitions

Abstract Type – Example

- Creates a new type (not equal to **int list**)
- Functional implementation of integer sets – **insert** creates new intset
- Exports type **intset**, **empty_set**, **insert**, **union**, **elt_of**, and **set_to_list**; act as primitive
 - Cannot use pattern matching or list functions; won't type check

Abstract Type – Example

- Implementation: just use record, except for type checking
- Data constructor **C** only visible inside the asbtype declaration; type rational visible outside
- Data abstraction allows us to prove data invariant holds for all objects of type rational: denominator never 0

Abstract Types

- A type is abstract if the user can only see the type, constants of that type (by name), and operations for interacting with objects of that type that have been explicitly exported
- Primitive types (eg. reals, integers) are built-in abstract types

User Defined Abstract Types

- Syntactic construct to provide encapsulation of abstract type implementation
- Inside, implementation visible to constants and subprograms
- Outside, only type name, not implementation, visible

Modules

- Language construct for grouping related types, data structures, and operations
- Typically allows at least some encapsulation
 - Can be used to provide abstract types

Modules

- Provides scope for variable and subprogram names
- Typically includes interface stating which modules it depends upon and what types and operations it exports
- Compilation unit for separate compilation

Modules

- Key differences between different languages:
 - Is a module itself a type or simply a collection of code?
 - Must a module declare what external types and functions it uses?
 - Can new modules be created from old ones (e.g., functors)?

Modules: Ada

- Modules called **packages**
- Two components: specification and **body** (both with same name)
- Components of module that are not exported are included in specification, in section called **private**
- No restriction on what types may be exported

Running Example – Stacks

ADA :

package STACKPACK is

 type STACKTYPE is limited private;

 MAX_SIZE : constant := 100;

 function EMPTY (STK : in
 STACKTYPE) return BOOLEAN;

Running Example – Stacks

```
procedure PUSH (STK : in out  
STACKTYPE;
```

```
        ELEMENT : in  
INTEGER);
```

```
procedure POP (STK : in out  
STACKTYPE);
```

```
function TOP (STK : in  
STACKTYPE) return INTEGER;
```

Running Example – Stacks

```
private
  type LIST_TYPE is array (1..MAX_SIZE)
  of INTEGER;
  type STACKTYPE is
  record
    LIST : LIST_TYPE;
    TOPSUB : INTEGER range
    0..MAX_SIZE := 0;
  end record;
end STACKPACK;
```

Running Example – Stacks

```
with TEXT_IO; use TEXT_IO;
package body STACKPACK is
  function EMPTY (STK : in STACKTYPE)
  return BOOLEAN is
  begin
    return STK.TOPSUB = 0
  end EMPTY;
```

Running Example – Stacks

```
procedure PUSH (STK : in out
STACKTYPE;
                ELEMENT : in INTEGER) is
begin
  if STK.TOPSUB >= MAX_SIZE
  then
    PUT_LINE ("ERROR - Stack
overflow");
```

Running Example – Stacks

```
else
    STK.TOPSUB := STK.TOPSUB
+ 1;
    STK.LIST(TOPSUB) :=
ELEMENT;
end if;
end PUSH;
```

Running Example – Stacks

```
procedure POP (STK : in out
STACKTYPE) is
begin
  if STK.TOPSUB = 0
    then PUT_LINE ("ERROR - Stack
underflow")
    else STK.TOPSUB := STK.TOPSUB - 1;
  end if;
end POP;
```

Running Example – Stacks

```
function TOP (STK : in STACKTYPE)
return INTEGER is
begin
  if STK.TOPSUB = 0
    then PUT_LINE ("ERROR - Stack is
empty");
    else return STK.LIST(STK.TOPSUB);
  end if;
end TOP;
end STACKPACK;
```

Running Example – Stacks

```
with STACKPACK, TEXT_IO;  
use STACKPACK, TEXT_IO;  
procedure USE_STACKS is  
  TOPONE : INTEGER;  
  STACK : STACKTYPE;  
begin
```

Running Example – Stacks

...

```
PUSH (STACK, 42);
```

```
PUSH (STACK, 27);
```

```
POP (STACK)
```

```
TOPONE := TOP (STACK);
```

...

```
end USE_STACKS;
```

Parameterized Modules

- With simple modules, can create module for stacks of integers of size 100, for example
- Would like to have module for use with any type
- Would like to have module for use with any size
- Want to pass type and value as arguments to module
- Done by parametrized modules

Running Example – Stacks

Ada:

generic

MAX_SIZE : POSITIVE;

type ELEMENT_TYPE is private;

package GENERIC_STACK is

Running Example – Stacks

type STACKTYPE is limited private;

function EMPTY (STK : in
STACKTYPE) return BOOLEAN;

...

function TOP (STK : in STACKTYPE)
return ELEMENT_TPYE;

Running Example – Stacks

```
private
  type LIST_TYPE is array
    (1..MAX_SIZE) of ELEMENT_TYPE;
  type STACKTYPE is
    record
      LIST : LIST_TYPE;
      TOPSUB : INTEGER range
        0..MAX_SIZE := 0;
    end record;
end GENERIC_STACK;
```

Running Example – Stacks

```
with TEXT_IO; use TEXT_IO;
package body GENERIC_STACK is
  function EMPTY (STK : in STACKTYPE)
  return BOOLEAN is
  begin
    return STK.TOPSUB = 0
  end EMPTY;
```

Running Example – Stacks

```
procedure PUSH (STK : in out
STACKTYPE;
                ELEMENT : in
ELEMENT_TYPE) is
begin
  if STK.TOPSUB >= MAX_SIZE
  then
    PUT_LINE ("ERROR - Stack
overflow");
```

Running Example – Stacks

```
else
  STK.TOPSUB := STK.TOPSUB + 1;
  STK.LIST(TOPSUB) := ELEMENT;
end if;
end PUSH;

...
end STACKPACK;
```

Running Example – Stacks

```
package INTEGER_STACK is new  
  GENERIC_STACK (100, INTEGER);
```

```
package FLOAT_STACK is new  
  GENERIC_STACK (500, FLOAT);
```

Object-Oriented Programming

- Class is a form of abstract type
- Instances of class called **objects**
- Operators of class called **methods**
- Applying a **method** to an object in the class called **message passing**
 - Messages take object as implicit argument

Variables and Methods

- Classes have two kinds of methods and two kinds of variable:
 - Instance methods and instance variables
 - Instance variables hold state of single object
 - Different objects have different (copies of) instance variables
 - Instance methods act on instance variables
 - Class methods and class variables
 - Class variables hold state that is shared in common among all objects of class

Inheritance

- **Inheritance** allows all variables and methods exported from **parent** class to be part of a **derived class** or **subclass**
- Export information:
 - **public**: everybody sees
 - **private**: only seen inside class
 - **protected**: exported only to subclasses
- If a method is added to a subclass with the same name (and usually signature) as in the parent class the new version **overrides** inherited version in subclass

Inheritance Polymorphism and Dynamic Binding

- Method of superclass may be overridden in subclass
- **Dynamic binding:** when a method of superclass is applied to object of subclass, methods of subclass are used in computing result, instead of methods of superclass
- Java has dynamic binding by default
- C++ method must be marked as virtual for dynamic binding to be used with it

Virtual (or Abstract) Methods

- **Virtual method:** has declaration but no function body
- Subclass must override all virtual methods with methods with fully concrete bodies to be instantiated

Single Versus Multiple Inheritance

- Single inheritance: Class may only be a subclass of one parent
- Multiple inheritance: Class may be immediate subclass of two or more parents
- Problem with multiple inheritance:

```
class A:B,C {...}
```

What if we have both B.m and C.m? Which method is A.m?

Answer in most languages: B.m

Infinite Data and Evaluation Methods

- Handling infinite data calls for different evaluation methods
- Three methods discussed below:
 - *Call by Value*
 - *Call by Name*
 - *Call by Need*

Call by Value

- Remember *eager evaluation*?
- Lambda calculus:
 - $(\lambda x.x)((\lambda y.y)z) \rightarrow (\lambda x.x)(z) \rightarrow z$
- This is call by value. It's what you are probably used to.
- OCaml, C/C++, Java, etc...

Call by Value Example

- let $f\ x = (x+1)^*(x+1)$
- $f\ ((3*4)/6) \rightarrow ?$

Call by Value Example

- let $f\ x = (x+1)^*(x+1)$
- $f\ ((3*4)/6)$
- $\rightarrow f\ (12/6)$

Call by Value Example

- $\text{let } f \ x = (x+1)^*(x+1)$
- $f \ ((3*4)/6)$
- $\rightarrow f \ (12/6)$
- $\rightarrow f \ 2$

Call by Value Example

- $\text{let } f \ x = (x+1)^*(x+1)$
- $f \ ((3*4)/6)$
- $\rightarrow f \ (12/6)$
- $\rightarrow f \ 2$
- $\rightarrow (2+1)^*(2+1)$

Call by Value Example

- let $f\ x = (x+1)^*(x+1)$
- $f\ ((3*4)/6)$
- $\rightarrow f\ (12/6)$
- $\rightarrow f\ 2$
- $\rightarrow (2+1)^*(2+1)$
- $\rightarrow 3^*(2+1)$

Call by Value Example

- let $f\ x = (x+1)^*(x+1)$
- $f\ ((3*4)/6)$
- $\rightarrow f\ (12/6)$
- $\rightarrow f\ 2$
- $\rightarrow (2+1)^*(2+1)$
- $\rightarrow 3^*(2+1)$
- $\rightarrow 3*3$

Call by Value Example

- $\text{let } f \ x = (x+1)^*(x+1)$
- $f \ ((3*4)/6)$
- $\rightarrow f \ (12/6)$
- $\rightarrow f \ 2$
- $\rightarrow (2+1)^*(2+1)$
- $\rightarrow 3^*(2+1)$
- $\rightarrow 3*3$
- $\rightarrow 9$

Call by Value Example

- $\text{let } f \ x = (x+1)^*(x+1)$

- $f \ ((3*4)/6)$

- $\rightarrow f \ (12/6)$

- $\rightarrow f \ 2$


- $\rightarrow (2+1)^*(2+1)$

- $\rightarrow 3*(2+1)$

- $\rightarrow 3*3$

- $\rightarrow 9$

Argument
evaluated *before*
function call



Only *values* are passed...
Not expressions.

Using Call by Value in Ocaml

[This slide left
intentionally blank]*

*Ocaml already uses *Call by Value* semantics,
so you don't need to do anything

Call by Value pros/cons

- The Good
 - Efficiency
 - Easy to implement
 - When side-effects are present, easier to understand effects of function call
- The Bad
 - Sometimes wasteful
- The Ugly
 - Can cause otherwise avoidable nontermination

Call by Value nontermination

- `let rec foo x = foo (x + 1);;`
 - `let fTrue a b = a;;`
 - `fTrue 5 (foo 10);;`
-
- We've seen this kind of thing before
 - (Eager evaluation in λ -calculus)

Call by Name

- Remember *lazy evaluation*?
- Lambda calculus:
 - $(\lambda x.x)((\lambda y.y)z) \rightarrow (\lambda y.y)z \rightarrow z$
- This is *Call by Name*
- On function call, pass in whole argument, without evaluating

Call by Name Example

- let $f\ x = (x+1)*(x+1)$
- $f\ ((3*4)/6) \rightarrow ?$

Call by Name Example

- $\text{let } f \ x = (x+1)^*(x+1)$
- $f \ ((3*4)/6)$
- $\rightarrow (((3*4)/6)+1)^*(((3*4)/6)+1)$

Call by Name Example

- $\text{let } f \ x = (x+1)^*(x+1)$
- $f \ ((3*4)/6)$
- $\rightarrow (((3*4)/6)+1)^*(((3*4)/6)+1)$
- $\rightarrow ((12/6)+1)^*(((3*4)/6)+1)$

Call by Name Example

- $\text{let } f \ x = (x+1)^*(x+1)$
- $f \ ((3*4)/6)$
- $\rightarrow (((3*4)/6)+1)^*(((3*4)/6)+1)$
- $\rightarrow ((12/6)+1)^*(((3*4)/6)+1)$
- $\rightarrow (2+1)^*(((3*4)/6)+1)$

Call by Name Example

- $\text{let } f \ x = (x+1)*(x+1)$
- $f \ ((3*4)/6)$
- $\rightarrow (((3*4)/6)+1)*(((3*4)/6)+1)$
- $\rightarrow ((12/6)+1)*(((3*4)/6)+1)$
- $\rightarrow (2+1)*(((3*4)/6)+1)$
- $\rightarrow (3)*(((3*4)/6)+1)$

Call by Name Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- $\rightarrow (((3*4)/6)+1)*(((3*4)/6)+1)$
- $\rightarrow ((12/6)+1)*(((3*4)/6)+1)$
- $\rightarrow (2+1)*(((3*4)/6)+1)$
- $\rightarrow (3)*(((3*4)/6)+1)$
- $\rightarrow (3)*((12/6)+1)$
- $\rightarrow (3)*(2+1) \rightarrow 3*3 \rightarrow 9$

Argument
passed *without*
evaluating



Using Call by Name in Ocaml

- Can *lambda lift* expressions to delay their evaluation (`fun () -> e`)

- Instead of

```
let f x = (x+1)*(x+1)
```

```
f ((3*4)/6)
```

- Write

```
let f x = ((x())+1)*((x())+1)
```

```
f (fun () -> ((3*4)/6))
```

Call by Name pros/cons

- The Good
 - If it is possible to terminate, it will.
- The Bad
 - Inefficient if you use argument more than once
 - Doesn't play nice with Side Effects

Side Effects

- Consider this C-like imperative code:

```
int x=5;
```

```
int square(int a)
```

```
{
```

```
    return a*a;
```

```
}
```

What does `square(++x)` return?

Side Effects

- What does `square(++x)` return?
- With *Call by Value*
 - Returns $6 * 6 = 36$
- With *Call by Name*
 - Returns $6 * 7 = 42$

But...

- The most useful property of *Call by Name* is its termination property.
 - If you have a nonterminating argument that **is not used**, then *Call by Name* will terminate and *Call by Value* won't.
 - Related: if you have an expensive argument that **is not used**, then *Call by Name* can be more efficient than *Call by Value*.

Call by Need

- The “Truly Lazy” form of Evaluation
 - *Call by Name* can end up doing more work than *Call by Value*, but *Call by Need* won't.
- Postpone argument evaluation until value actually needed, *but* only evaluate once.
 - Have to use side-effects!

Call by Need Example

- let $f\ x = (x+1)^*(x+1)$
- $f\ ((3*4)/6) \rightarrow ?$

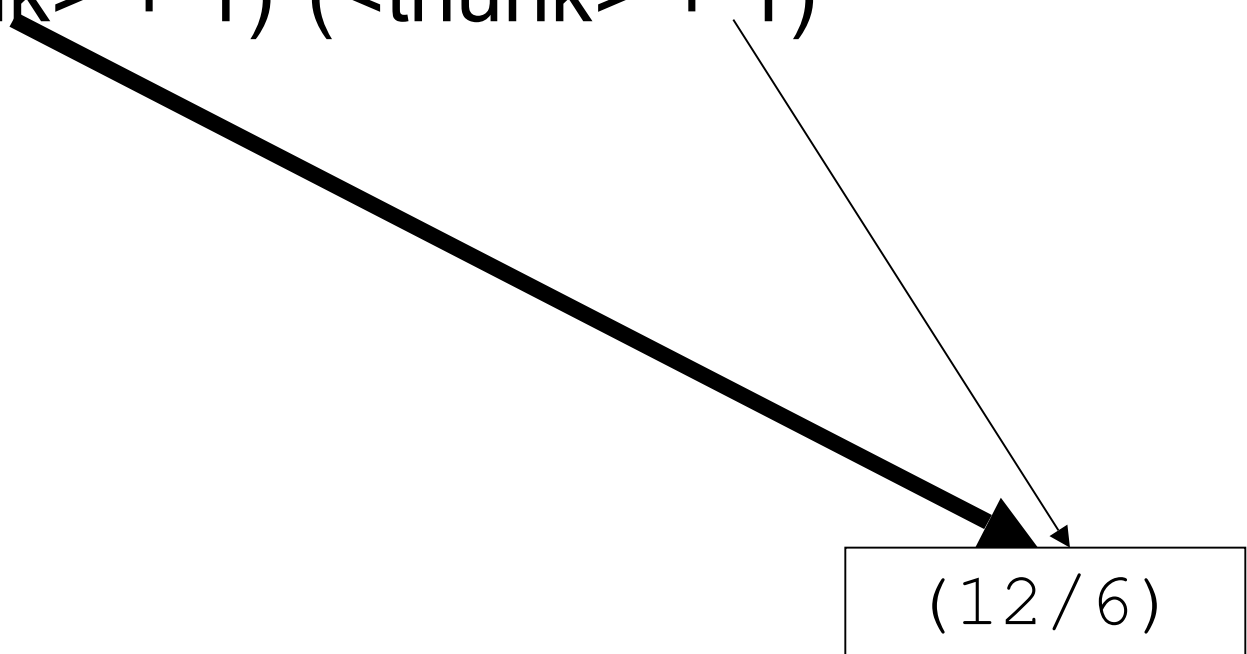
Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- \rightarrow (<thunk> + 1)*(<thunk> + 1)

$((3*4)/6)$

Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- \rightarrow (<thunk> + 1)*(<thunk> + 1)



(12 / 6)

Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- \rightarrow (<thunk> + 1)*(<thunk> + 1)

2

Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- \rightarrow (<thunk> + 1)*(<thunk> + 1)
- \rightarrow (2 + 1) * (<thunk> + 1)



Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- $\rightarrow (\langle\text{thunk}\rangle + 1) * (\langle\text{thunk}\rangle + 1)$
- $\rightarrow (2 + 1) * (\langle\text{thunk}\rangle + 1)$
- $\rightarrow 3 * (\langle\text{thunk}\rangle + 1)$



2

Call by Need Example

- $\text{let } f \ x = (x+1)^*(x+1)$
- $f \ ((3*4)/6)$
- $\rightarrow (\langle\text{thunk}\rangle + 1)^*(\langle\text{thunk}\rangle + 1)$
- $\rightarrow (2 + 1) * (\langle\text{thunk}\rangle + 1)$
- $\rightarrow 3 * (\langle\text{thunk}\rangle + 1)$
- $\rightarrow 3 * (2 + 1)$

Call by Need Example

- let f x = (x+1)*(x+1)
- f ((3*4)/6)
- \rightarrow (<thunk> + 1)*(<thunk> + 1)
- \rightarrow (2 + 1) * (<thunk> + 1)
- \rightarrow 3 * (<thunk> + 1)
- \rightarrow 3 * (2 + 1) *Argument is not*
- \rightarrow 3 * 3 *evaluated until it*
- \rightarrow 9 *is needed, but it*
- is evaluated at*
- most once*

2

Side-Effects in Ocaml

- Ocaml has some imperative features

```
# let counter = ref 0;;
```

```
val counter : int ref = {contents = 0}
```

```
# counter := !counter + 1;;
```

```
# !counter;;
```

```
- : int = 1
```

```
# counter := !counter + 1;;
```

```
# !counter;;
```

```
- : int = 2
```

Implementing Thunks

- We need three things:
 - A *thunk* or *suspension* to hold the data
 - A function *delay* to suspend an expression
 - A function *force* to give us the value of the suspended expression

```
# type 'a thunk_type =  
  Value of 'a  
| Susp of (unit -> 'a);;
```

Implementing Thunks

```
# let delay f =  
  let thunk = ref (Susp f) in  
  fun () -> match (!thunk) with  
  | Value a -> a  
  | Susp f ->  
    let result = f () in  
    (thunk := (Value result));  
    result );;
```

Implementing Thunks

```
# let force f = f ();;
```

- Instead of

```
let f x = (x+1)*(x+1)
```

```
f ((3*4)/6)
```

- Write

```
let f x = ((force x)+1)*((force x)+1)
```

```
f (delay (fun () -> ((3*4)/6)))
```

The Lazy Module

- Actually, this has been implemented for you already in Ocaml's "Lazy" module
- `lazy` creates the suspension (thunk)
- `Lazy.force` gets the value

```
let f x = ((Lazy.force x)+1)
          *((Lazy.force x)+1)
f (lazy ((3*4)/6))
```


Infinite Data

- `let rec ones = 1::ones`
- But how would you operate on that data?
 - For example, what if you want to do `map` on `ones`?

```
# let twos=List.map ((+) 1) ones;;
```

Stack overflow during evaluation (looping recursion?).

Lazy Lists

```
type 'a llist =
```

```
  Cons of 'a * 'a llist Lazy.t
```

```
  | Nil;;
```

```
let rec ftake n llist =
```

```
  match n, llist with
```

```
  | _, Nil -> []
```

```
  | 0, _ -> []
```

```
  | _, (Cons(x, xs)) -> x :: ftake (n-1) (Lazy.force xs);;
```

Infinite Data the Lazy way

```
# let rec ones = Cons(1, lazy ones);;  
val ones : int list = Cons (1, <lazy>)
```

```
# ftake 30 ones;;
```

```
- : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;  
1; 1; 1; 1; 1; 1; 1; 1; 1]
```

Map on an Infinite List

```
let rec lmap f llst =  
  match llst with  
  | Cons (x,xs) -> Cons (f x,  
    lazy(lmap f (Lazy.force xs)))  
  | Nil -> Nil
```

Map on an Infinite List

```
# let twos = lmap ((+) 1) ones;;  
val twos : int llist = Cons (2, <lazy>)  
# ftake 30 twos;;  
- : int list =  
[2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2; 2;  
 2; 2; 2; 2; 2; 2; 2; 2]
```

The List of Natural Numbers

let rec nats = ?

Hint: You're going to use `lmap`

The List of Natural Numbers

```
# let rec nats = Cons(1, lazy (lmap ((+) 1)
  nats));;
val nats : int list = Cons (1, <lazy>)
# ftake 30 nats;;
- : int list =
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15;
 16; 17; 18; 19; 20; 21; 22; 23; 24; 25; 26; 27;
 28; 29; 30]
```

Question

```
# let plus a b = print_string "Plus "; a+b;;  
val plus : int -> int -> int = <fun>  
# let rec nats = Cons(1, lazy (lmap (plus 1) nats));;  
val nats : int llist = Cons (1, <lazy>)  
# ftake 3 nats;;  
Plus Plus Plus- : int list = [1; 2; 3]  
# ftake 5 nats;;  
How many times is Plus printed here?
```

Answer

```
# let plus a b = print_string "Plus "; a+b;;  
val plus : int -> int -> int = <fun>  
# let rec nats = Cons(1, lazy (lmap (plus 1) nats));;  
val nats : int llist = Cons (1, <lazy>)  
# ftake 3 nats;;  
Plus Plus Plus- : int list = [1; 2; 3]  
# ftake 5 nats;;  
Plus Plus- : int list = [1; 2; 3; 4; 5]
```