

Compiler Construction Lent Term 2015

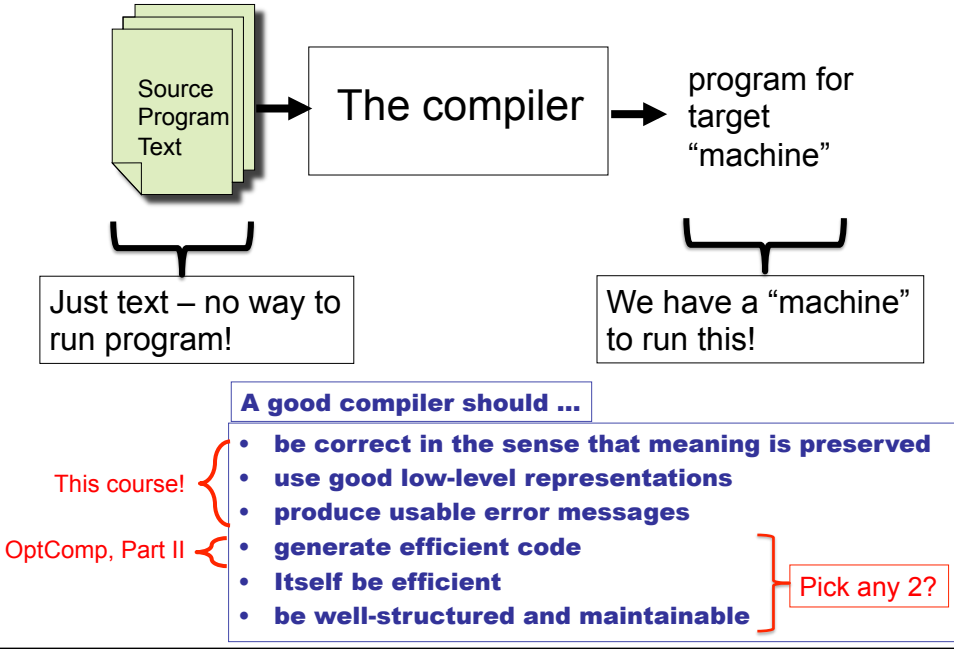
Lectures 1 - 4 (of 16)

Timothy G. Griffin
tgg22@cam.ac.uk

Computer Laboratory
University of Cambridge

1

Compilation is a special kind of translation



Why Study Compilers?

- **Although many of the basic ideas were developed over 40 years ago, compiler construction is still an evolving and active area of research and development.**
- **Compilers are intimately related to programming language design and evolution.**
- **Compilers are a Computer Science success story illustrating the hallmarks of our field --- higher-level abstractions implemented with lower-level abstractions.**
- **Every Computer Scientist should have a basic understanding of how compilers work.**

3

Mind The Gap

High Level Language

- Machine independent
- Complex syntax
- Complex type system
- Variables
- Nested scope
- Procedures, functions
- Objects
- Modules
- ...

Typical Target Language

- Machine specific
- Simple syntax
- Simple types
- memory, registers, words
- Single flat scope

Help!!! Where do we begin???

4

The Gap, illustrated

```
public class Fibonacci {
    public static long fib(int m) {
        if (m == 0) return 1;
        else if (m == 1) return 1;
        else return
            fib(m - 1) + fib(m - 2);
    }
    public static void
    main(String[] args) {
        int m =
            Integer.parseInt(args[0]);
        System.out.println(
            fib(m) + "\n");
    }
}
```

javac Fibonacci.java
javap -c Fibonacci.class

```
public class Fibonacci {
    public Fibonacci();
    Code:
    0: aload_0
    1: invokespecial #1
    4: return
    public static long fib(int);
    Code:
    0: iload_0
    1: ifne        6
    4: lconst_1
    5: lreturn
    6: iload_0
    7: lconst_1
    8: if_icmpne  13
    11: lconst_1
    12: lreturn
    13: iload_0
    14: lconst_1
    15: isub
    16: invokestatic #2
    19: iload_0
    20: lconst_2
    21: isub
    22: invokestatic #2
    25: ladd
    26: lreturn
}
```

```
public static void
main(java.lang.String[]);
Code:
0: aload_0
1: lconst_0
2: aaload
3: invokestatic #3
6: istore_1
7: getstatic  #4
10: new        #5
13: dup
14: invokespecial #6
17: iload_1
18: invokestatic #2
21: invokevirtual #7
24: ldc      #8
26: invokevirtual #9
29: invokevirtual #10
32: invokevirtual #11
35: return
}
```

JVM bytecodes

5

The Gap, illustrated

fib.ml

```
(* fib : int -> int *)
let rec fib m =
  if m = 0
  then 1
  else if m = 1
  then 1
  else fib(m - 1) + fib(m - 2)
```

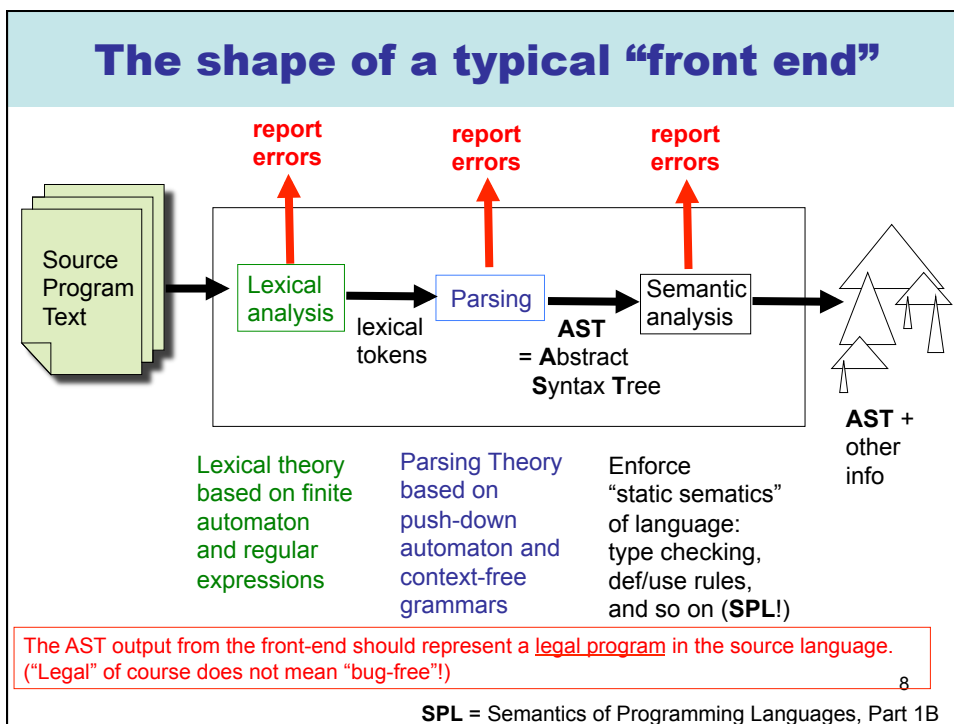
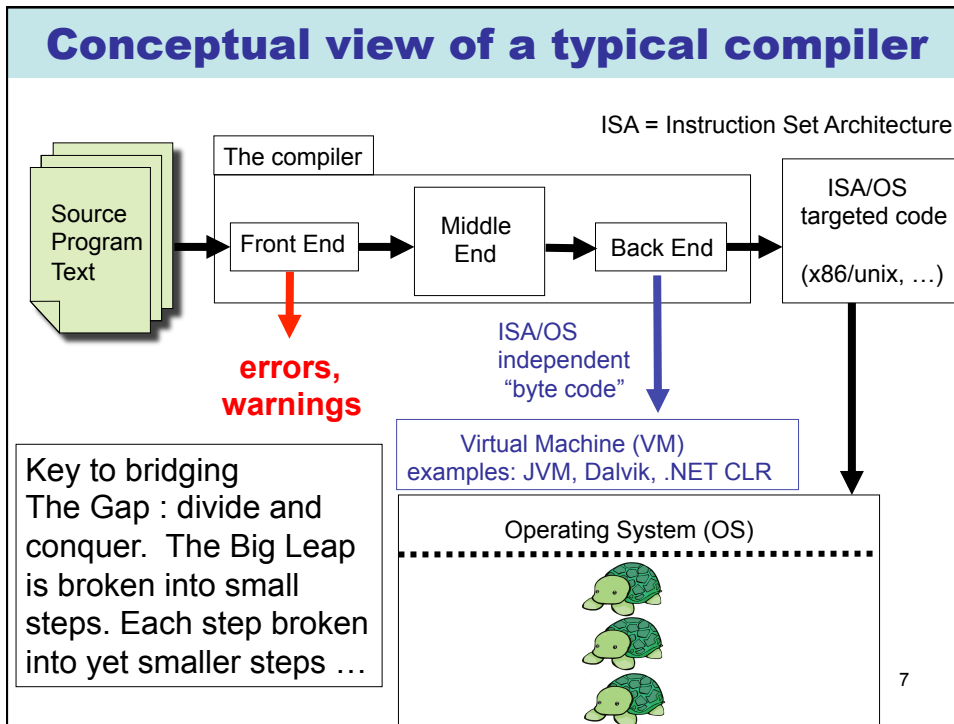
ocamlc -dinstr fib.ml

```
L1:    branch L2
      acc 0
      push
      const 0
      eqint
      branchifnot L4
      const 1
      return 1
L4:    acc 0
      push
      const 1
      eqint
      branchifnot L3
      const 1
      return 1
```

```
L3:    acc 0
      offsetint -2
      push
      offsetclosure 0
      apply 1
      push
      acc 1
      offsetint -1
      push
      offsetclosure 0
      apply 1
      addint
      return 1
L2:    clouserrec 1, 0
      acc 0
      makeblock 1, 0
      pop 1
      setglobal Fib!
```

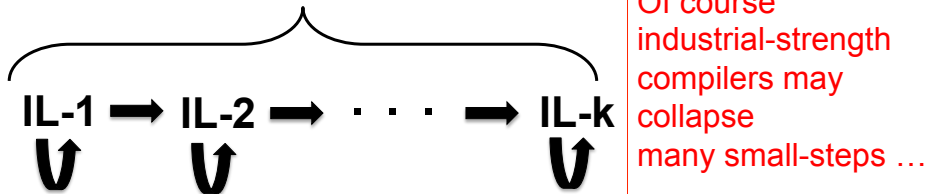
OCaml VM bytecodes

6



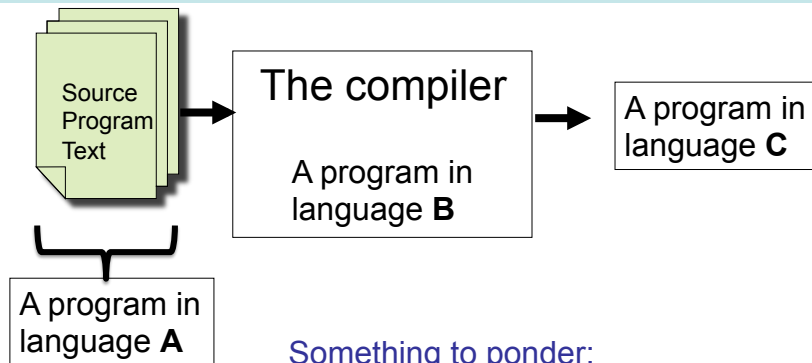
Our view of the middle- and back-ends : a sequence of small transformations

Intermediate Languages



- Each **IL** has its own semantics (perhaps informal)
- Each transformation (\Rightarrow) preserves semantics (**SPL!**)
- Each transformation eliminates only a few aspects of **the gap**
- Each transformation is fairly easy to understand
- Some transformations can be described as “optimizations”
- We will associate each **IL** with its own interpreter/VM. (Again, not something typically done in “industrial-strength” compilers.)

Compilers must be compiled



Something to ponder:

A compiler is just a program.

But how did it get compiled?

The OCaml compiler is written in OCaml.

How was the compiler compiled?

Approach Taken

- We will develop compilers for fragments of the languages introduced in Semantics of Programming Languages, Part 1B.
- We will pay special attention to the correctness of our compilers.
- We will compile only to Virtual Machines (VMs) of various kinds. See Part II optimising compilers for generating lower-level code.
- The toy compilers and some VMs will be available on the course web site.
- We will be using the OCaml dialect of ML.

11

OCaml

- Install from <https://ocaml.org>.
- See OCaml Labs : <http://www.cl.cam.ac.uk/projects/ocaml/labs>.
- A side-by-side comparison of SML and OCaml Syntax: <http://www.mpi-sws.org/~rossberg/sml-vs-ocaml.html>

- Download from the course website
 - basic_transformations.tar.gz
 - slang1_interpret.tar.gz
 - Build with “ocamlbuild slang.byte”

12

SML Syntax

vs.

OCaml Syntax

```
datatype 'a tree =  
  Leaf of 'a  
  | Node of 'a * ('a tree) * ('a tree)  
  
fun map_tree f (Leaf a) = Leaf (f a)  
  | map_tree f (Node (a, left, right)) =  
    Node(f a, map_tree f left, map_tree f right)  
  
val map_list = map_tree (fn a => [a])
```

```
type 'a tree =  
  Leaf of 'a  
  | Node of 'a * ('a tree) * ('a tree)  
  
let rec map_tree f = function  
  | Leaf a -> Leaf (f a)  
  | Node (a, left, right) ->  
    Node(f a, map_tree f left, map_tree f right)  
  
let map_list = map_tree (fun a -> [a])
```

For more examples see my [sml_vs_ocaml.ml](#) on the course website.

13

The Shape of this Course

1. Overview
2. Slang.1. Front-end, High-level interpreter
3. Eliminating recursion I
4. Eliminating recursion II
5. Deriving the Slang.1 VM-1
6. Deriving the Slang.1 VM-1
7. Deriving the Slang.1 VM-2
8. Deriving the Slang.1 VM-2, with some optimisations
9. Slang.2 : higher order functions
10. Slang.2 : higher order functions, objects
11. Heap allocation, garbage collection
12. Assorted topics : bootstrapping a compiler, compilation units, linking
13. Lexical analysis : application of Theory of Regular Languages and Finite Automata
14. Generating Recursive descent parsers
15. Beyond Recursive Descent Parsing I
16. Beyond Recursive Descent Parsing II

LECTURE 2

Slang1. Front End

- Slang (= Simple LANGUAGE)
- Slang.1 : syntax, types, semantics
- The Front End
- A high-level interpreter for Slang.1 in Ocaml

15

Slang.1 examples

slang1_interpret/examples/fib.slang

```
let fib( m : int) : int =
  if m = 0
  then 1
  else if m = 1
  then 1
  else fib (m - 1) +
        fib (m - 2)
in
  fib(?)
end
```

The ? requests an integer input from the terminal

slang1_interpret/examples/gcd.slang

```
let gcd( m : int, n : int) : int =
  if m = n
  then m
  else if m < n
  then gcd(m, n - m)
  else gcd(m - n, n)
in
  let x : int = ?
  and y : int = ?
  in
    gcd(x, y)
  end
end
```

16

Slang.1 Front End

Input file



Parse (we use a version of LEX and YACC, which are covered in Lectures 13 -- 16).

Past.expr



Static analysis : checks types, and context-sensitive rules (no duplicate argument/let identifiers in let declaration, etc). Determine which functions are recursive, which = is used.

Past.expr



Eliminate "syntactic sugar"

Ast.exp



"Alpha convert" to ensure all bound variables are unique. In this way we will never have to worry about name clashes. This approach is a bit more "debugger friendly" than

Ast.exp

17

slang.byte demo in Lecture ...

Usage: slang.byte [options] [<file>]

Options are:

- V verbose front end
- v verbose interpreter(s)
- i0 Interpreter 0 (definitional interpreter)
- t run all test/*.slang with each selected interpreter, report unexpected outputs (silent otherwise)
- help Display this list of options
- help Display this list of options

18

Slang.1 Syntax (somewhat informal)

op ::= + | - | * | < | = | && | ||

t ::= bool | int | unit

e ::= ()
| n
| ? (*? requests an integer input from terminal*)
| x
| true
| false
| ~e (*boolean negation*)
| -e (*integer negation*)
| (e)
| (e op e)
| if e then else e
| let x : t = e in e end
| let x1 : t1 = e1
and x2 : t2 = e2
and and xn : tn = en
in e end
| let f (x1 : t1, ... , xn : tn) : t = e in e end
| f(e1, ... en)

19

Slang.1 Types and Semantics

Slang.1 is a simplified version of L2 from
Semantics of Programming Languages, Part 1B.

- we have added input (?) and additional primitive operations
- we have simplified the concrete syntax
- we have restricted functions to first-order functions

See Semantics notes for typing rules and operational semantics.

20

Parsed AST (past.ml)

```
type var = string

type type_expr = TEint | TEbool | TEunit

type formals = (var * type_expr) list

type oper = ADD | MUL | SUB | LT | AND | OR | EQ | EQB | EQI

type unary_oper = NEG | NOT

type loc = Lexing.position

type expr =
  | Unit of loc
  | What of loc
  | Var of loc * var
  | Integer of loc * int
  | Boolean of loc * bool
  | UnaryOp of loc * unary_oper * expr
  | Op of loc * expr * oper * expr
  | If of loc * expr * expr * expr
  | App of loc * var * expr list
  | Let of loc * binding_list * expr
  | LetFun of loc * var * formals * type_expr * expr * expr
  | LetRecFun of loc * var * formals * type_expr * expr * expr

and binding_list = (var * type_expr * expr) list
```

Locations (loc) are used in generating error messages from the front end.

Only the **LetFun** construct is Returned by the parser. The front end determines which declarations are recursive and replaces **LetFun** with **LetRecFun**

21

Internal AST (ast.ml)

```
type var = string

type formals = var list

type expr =
  | Unit
  | Var of var
  | Integer of int
  | Boolean of bool
  | If of expr * expr * expr
  | App of var * expr list
  | LetFun of var * formals * expr * expr
  | LetRecFun of var * formals * expr * expr
```

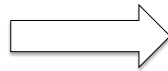
The internal AST (output of the front end) is simpler than the parsed AST. It

- eliminates types (is this really a good idea?)
- eliminates simple "lets"
- eliminates locations (loc)
- eliminates ? and all unary and binary operations (replaces them with function calls to "build-in" functions).

22

The “let” transform

```
let x1 = e1
and x2 = e2
and ...
and xn = en
in e end
```



```
let f(x1, x2, ..., xn) = e
in
  f(e1, e2, ..., en)
end
```

Where f is a fresh variable.

This is done to simplify some of our code.
Is it a good idea? Perhaps not.

23

slang.byte -V examples/fib.slang

Parsed result:

```
let fib(m : int) : int =
if (m = 0) then 1 else if (m = 1) then 1 else (fib((m - 1)) + fib((m - 2)))
in fib(?) end
```

After static check :

```
letrec fib(m : int) : int =
if (m = 0) then 1 else if (m = 1) then 1 else (fib((m - 1)) + fib((m - 2)))
in fib(?) end
```

After `Past_to_ast.translate_expr` :

```
letrec fib(m) =
if _eqi(m, 0) then 1 else if _eqi(m, 1) then 1 else _plus(fib(_subt(m, 1)), fib(_subt(m, 2)))
in fib(_read(())) end
```

After `Alpha.convert` :

```
letrec fib(m) =
if _eqi(m, 0) then 1 else if _eqi(m, 1) then 1 else _plus(fib(_subt(m, 1)), fib(_subt(m, 2)))
in fib(_read(())) end
```

24

slang.byte -V tests/alpha.slang

```
Parsed result:
let x : int = 1 in
let x : int = (2 + x) in
let x : int = (3 + x) in
let g(x : int) : int = (x + x) in
let h(x : int) : int = (x + g(x)) in
  g(h(g(x))) end end end end end
```

... ..

```
After Alpha.convert :
let _0(x) =
  let _1(_x) =
    let _2(__x) =
      let g(__x) = _plus(__x, __x)
      in let h(__x) = _plus(__x, g(__x))
        in g(h(g(__x))) end
      end
    in _2(_plus(3, _x)) end
  in _1(_plus(2, x)) end
in _0(1) end
```

OK, this is not
so pretty ...

25

common.mli

```
exception Error of string

type constant =
| INT of int
| BOOL of bool
| UNIT

val complain : string -> 'a

val string_of_constant : constant -> string

val bool_of_constant : constant -> bool

....

....
```

Basic “run time” constants.
These will be used
by multiple interpreters
and VMs

26

The Interpreter! interp_0.mli

```
type basic_value =
  | SIMPLE of Common.constant
  | TUPLE of Common.constant list

type value =
  | BASIC of basic_value
  | FUN of (basic_value -> Common.constant)

type env = Ast.var -> value
type state = env * Ast.expr
type binding = Ast.var * value
type bindings = binding list

val constant_of_value : value -> Common.constant
val function_of_value : value -> (basic_value -> Common.constant)
val update : (env * binding) -> env
val bind_args : (env * Ast.formals * basic_value) -> env

val eval : state -> Common.constant
val eval_args : (env * (Ast.expr list)) -> Common.constant list
val interpret : Ast.expr -> Common.constant
```

27

Interp_0.eval

```
let rec eval (env, e) =
  match e with
  | Unit      -> UNIT
  | Var x     -> gs (env x)
  | Integer n -> INT n
  | Boolean b -> BOOL b
  | If(e1, e2, e3) -> if gb(eval(env, e1)) then eval(env, e2) else eval(env, e3)
  | App(f, [e]) -> (gf (env f)) (SIMPLE(eval(env, e)))
  | App(f, el) -> (gf (env f)) (TUPLE(eval_args(env, el)))
  | LetFun(f, fl, e1, e2) ->
    let new_env = update(env, (f, FUN (fun v -> eval(bind_args(env, fl, v), e1))))
    in eval(new_env, e2)
  | LetRecFun(f, fl, e1, e2) ->
    let rec new_env g = (* Note the recursive environment! *)
      if g = f then FUN (fun v -> eval(bind_args(new_env, fl, v), e1)) else env g
    in eval(new_env, e2)

and eval_args(env, el) =
  match el with
  | [] -> []
  | e :: rest -> (eval(env, e)) :: (eval_args(env, rest))
```

28

Observations

- This could be called a “definitional interpreter” --- we are defining the semantics of Slang.1 (the defined language) in terms of high-level constructs of OCaml (the defining language).
- Note that Slang.1 functions are interpreted as OCaml functions, Slang.1 application as OCaml application.
- The only “tricky bit” involves recursive Slang.1 functions. Here we use a recursive definition in OCaml --- but in the definition of the environment. The body of a recursive function *f* must be able to find its own definition in the environment!

29

Are we done?

- **Our interpreter runs correct Slang programs**
- **It reports errors for badly constructed programs**
- **What more do we need?**
- **Class dismissed!**

- **Oh, wait a second ...**

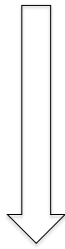
30

Where are we going?

Slang.1  VM code
compile to

The goal of first Lectures 3 – 8.

Interpreter 0



Slang-VM

We will derive our own Slang Virtual Machine (Slang-VM).

This derivation will be done, step-by-step, via semantics preserving transformations applied to the interpreter!

31

Derive? How?

Interpreter 0



Eliminate higher-order functions with “defunctionalisation” (DFC)

Interpreter 1



Replace recursion with iteration via the Continuation Passing Style (CPS) transformation.

Interpreter 2



Eliminate higher-order functions with “defunctionalisation” (DFC)

Interpreter 3



“Stackify” : represent defunctionalised continuations as a stack.

Interpreter 4

Derive? How?

Interpreter 4



Split single stack into three stacks.

Interpreter 5



Refactor. Compile expression to instructions!

Slang VM 1



“Optimise” and recombine stacks into one.

Slang VM 2

**Lectures 3 & 4 : introduction to basic techniques (cps, dfc)
Lectures 5 & 6 : Derive Slang VM1 from eval
Lectures 7 & 8 : Slang VM 2, and other optimisations.**

LECTURE 3 & 4 Eliminating Recursion

- Evaluation using a stack
- Recursion using a stack
- Tail recursion elimination: from recursion to iteration
- Continuation Passing Style (CPS) : transform any recursive function to a tail-recursive function
- “Defunctionalisation” (DFC) : replace higher-order functions with a data structure
- Putting it all together:
 - Derive the Fibonacci Machine
 - Derive the Expression Machine, and “compiler”!

34

Evaluation viewed as a sequence of operations on a stack

e1 op e2



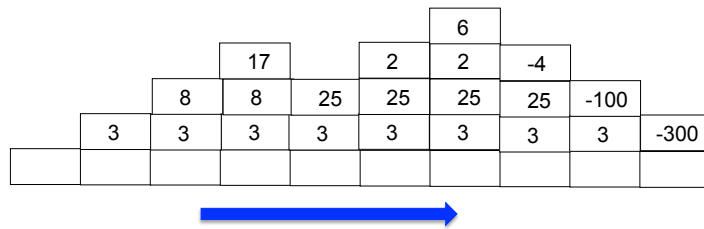
... code for e1...

... code for e2 ...

op

push 3
push 8
push 17
add
push 2
push 6
sub
mul
mul

$3 * ((8 + 17) * (2 - 6))$



Example : Fibonacci Numbers

```
(* fib : int -> int *)
let rec fib m =
  if m = 0
  then 1
  else if m = 1
  then 1
  else fib(m - 1) + fib (m - 2)
```

```
List.map fib [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;
```

```
= [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
```

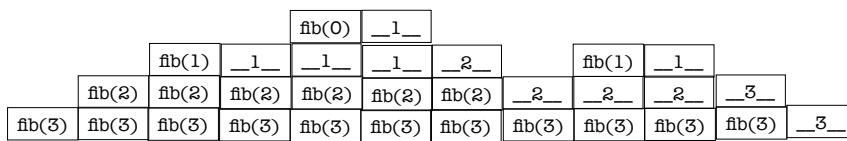
Fibonacci Numbers

```

let rec fib m =
  if m = 0
  then 1
  else if m = 1
  then 1
  else fib(m - 1) + fib (m - 2)
  
```

```

List.map fib [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;
= [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
  
```



This is a very abstract picture of what might be happening in the low-level stack-oriented Virtual Machine (VM) of OCaml

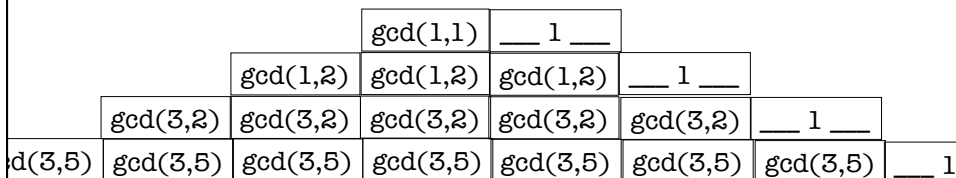
37

Example of tail-recursion : gcd

```

(* gcd : int * int -> int *)
let rec gcd(m, n) =
  if m = n
  then m
  else if m < n
  then gcd(m, n - m)
  else gcd(m - n, n)
  
```

Compared to fib, this function uses recursion in a different way. It is **tail-recursive**. If implemented with a stack, then the “call stack” (at least with respect to gcd) will simply grow and then shrink. No “ups and downs” in between.



Tail-recursive code can be replaced by iterative code that does not require a “call stack” (constant space)

38

gcd_iter : Look Mom, no recursion!

```
(* gcd : int * int -> int *)
let rec gcd(m, n) =
  if m = n
  then m
  else if m < n
       then gcd(m, n - m)
       else gcd(m - n, n)
```

Here we have illustrated tail-recursion elimination as a source-to-source transformation. However, the OCaml compiler will do something similar to a lower-level intermediate representation. **Upshot** : we will consider all tail-recursive OCaml functions as representing iterative programs.

```
(* gcd_iter : int * int -> int *)
let gcd_iter (m, n) =
  let rm = ref m
  in let rn = ref n
  in let result = ref 0
  in let not_done = ref true
  in let _ =
    while !not_done
    do
      if !rm = !rn
      then (not_done := false;
            result := !rm)
      else if !rm < !rn
            then rn := !rn - !rm
            else rm := !rm - !rn
    done
  in !result
```

Familiar examples : fold_left, fold_right

From ocaml-4.01.0/stdlib/list.ml :

```
(* fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
*)
fold_left f a [b1; ...; bn] = f (... (f (f a b1) b2) ...) bn
let rec fold_left f a l =
  match l with
  | [] -> a
  | b :: rest -> fold_left f (f a b) rest

(* fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
*)
fold_right f [a1; ...; an] b = f a1 (f a2 (... (f an b) ...))
let rec fold_right f l b =
  match l with
  | [] -> b
  | a :: rest -> f a (fold_right f rest b)
```

This is tail recursive

This is NOT tail recursive

40

Question: can we transform any recursive function into a tail recursive function?

The answer is **YES!**

- We add an extra argument, called a *continuation*, that represents “the rest of the computation”
- This is called the Continuation Passing Style (CPS) transformation.
- We will then “defunctionalize” (DFC) these continuations and represent them with a stack.
- **Finally, we obtain a tail recursive function that carries its own stack as an extra argument!**

Reminder : we will apply this kind of transformation to `Interp_0.eval` as the first steps towards deriving a VM.

41

(CPS) transformation of fib

```
(* fib : int -> int *)
let rec fib m =
  if m = 0
  then 1
  else if m = 1
       then 1
       else fib(m - 1) + fib (m - 2)

(* fib_cps : int * (int -> int) -> int *)
let rec fib_cps (m, cnt) =
  if m = 0
  then cnt 1
  else if m = 1
       then cnt 1
       else fib_cps(m - 1, fun a -> fib_cps(m - 2, fun b -> cnt (a + b)))
```

42

A closer look

The rest of the computation after computing “fib(m)”. That is, cnt is a function expecting the result of “fib(m)” as its argument.

```
let rec fib_cps (m, cnt) =  
  if m = 0  
  then cnt 1  
  else if m = 1  
    then cnt 1  
    else fib_cps(m - 1, fun a -> fib_cps(m - 2, fun b -> cnt (a + b)))
```

The computation waiting
for the result of “fib(m-1)”

The computation waiting
for the result of “fib(m-2)”

This makes explicit the order of evaluation that is implicit in the original “fib(m-1) + fib(m-2)” :
-- first compute fib(m-1)
-- then compute fib(m-1)
-- then add results together
-- then return

43

Expressed without “lambdas”

```
(* fib_cps_v2 : (int -> int) * int -> int *)  
let rec fib_cps_v2 (m, cnt) =  
  if m = 0  
  then cnt 1  
  else if m = 1  
    then cnt 1  
    else let cnt2 a b = cnt (a + b)  
         in let cnt1 a = fib_cps_v2(m - 2, cnt2 a)  
         in fib_cps_v2(m - 1, cnt1)
```

Some prefer writing CPS forms without explicit funs

44

Use the identity continuation ...

```
(* fib_cps : int * (int -> int) -> int *)
let rec fib_cps (m, cnt) =
  if m = 0
  then cnt 1
  else if m = 1
       then cnt 1
       else fib_cps(m - 1, fun a -> fib_cps(m - 2, fun b -> cnt (a + b)))

let id (x : int) = x

let fib_1 x = fib_cps(x, id)
```

```
List.map fib_1 [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;
= [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
```

45

Correctness?

For all $c : \text{int} \rightarrow \text{int}$, for all m , $0 \leq m$,
we have, $c(\text{fib } m) = \text{fib_cps}(m, c)$.

Proof: assume $c : \text{int} \rightarrow \text{int}$. By Induction
on m . Base case : $m = 0$:
 $\text{fib_cps}(0, c) = c(1) = c(\text{fib}(0))$.

Induction step: Assume for all $n < m$, $c(\text{fib } n) = \text{fib_cps}(n, c)$.
(That is, we need course-of-values induction!)

```
fib_cps(m + 1, c)
= if m + 1 = 1
  then c 1
  else fib_cps((m+1) - 1, fun a -> fib_cps((m+1) - 2, fun b -> c (a + b)))
= if m + 1 = 1
  then c 1
  else fib_cps(m, fun a -> fib_cps(m-1, fun b -> c (a + b)))
= (by induction)
  if m + 1 = 1
  then c 1
  else (fun a -> fib_cps(m - 1, fun b -> c (a + b))) (fib m)
```

46

Correctness?

```
= if m + 1 = 1
  then c 1
  else fib_cps(m-1, fun b -> c ((fib m) + b))
= (by induction)
  if m + 1 = 1
  then c 1
  else (fun b -> c ((fib m) + b)) (fib (m-1))
= if m + 1 = 1
  then c 1
  else c ((fib m) + (fib (m-1)))
= c (if m + 1 = 1
     then 1
     else ((fib m) + (fib (m-1))))
= c(if m + 1 = 1
    then 1
    else fib((m + 1) - 1) + fib ((m + 1) - 2))
= c (fib(m + 1))
```

QED.

47

fib_cps expressed without “lambdas”

```
(* fib_cps_v2 : (int -> int) * int -> int *)
let rec fib_cps_v2 (m, cnt) =
  if m = 0
  then cnt 1
  else if m = 1
       then cnt 1
       else let cnt2 a b = cnt (a + b)
            in let cnt1 a = fib_cps_v2(m - 2, cnt2 a)
            in fib_cps_v2(m - 1, cnt1)
```

Idea of defunctionalisation (DFC): replace `id`, `cnt1` and `cnt2` with instances of a new data type:

```
type cnt = ID | CNT1 of int * cnt | CNT2 of int * cnt
```

Now we need an “apply” function of type `cnt * int -> int`

48

“Defunctionalised” version of fib_cps

```
(* datatype to represent continuations *)
type cnt = ID | CNT1 of int * cnt | CNT2 of int * cnt

(* apply_cnt : cnt * int -> int *)
let rec apply_cnt = function
| (ID, a)           -> a
| (CNT1 (m, cnt), a) -> fib_cps_dfc(m - 2, CNT2 (a, cnt))
| (CNT2 (a, cnt), b) -> apply_cnt (cnt, a + b)

(* fib_cps_dfc : (cnt * int) -> int *)
and fib_cps_dfc (m, cnt) =
  if m = 0
  then apply_cnt(cnt, 1)
  else if m = 1
       then apply_cnt(cnt, 1)
       else fib_cps_dfc(m - 1, CNT1(m, cnt))

(* fib_2 : int -> int *)
let fib_2 m = fib_cps_dfc(m, ID)
```

49

Correctness?

Let [c] be of type cnt representing
a continuation $c : \text{int} \rightarrow \text{int}$ constructed by fib_cps.

Then

$\text{apply_cnt}([c], m) = c(m)$

and

$\text{fib_cps}(n, c) = \text{fib_cps_dfc}(n, [c])$.

$[\text{fun } a \rightarrow \text{fib_cps}(m - 2, \text{fun } b \rightarrow \text{cnt}(a + b))] = \text{CNT1}(m, [\text{cnt}])$

$[\text{fun } b \rightarrow \text{cnt}(a + b)] = \text{CNT2}(a, [\text{cnt}])$

$[\text{fun } x \rightarrow x] = \text{ID}$

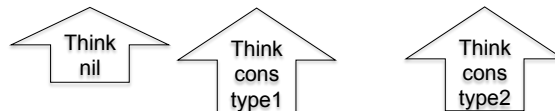
Proof left as an exercise!

50

Eureka! Continuations are just lists (used like a stack)

```
type int_list = NIL | CONS of int * int_list
```

```
type cnt = ID | CNT1 of int * cnt | CNT2 of int * cnt
```



Replace the above continuations with lists! (I've selected more suggestive names for the constructors.)

```
type tag = SUB2 of int | PLUS of int
type tag_list_cnt = tag list
```

51

Use a stack (implemented with a list)

```
type tag = SUB2 of int | PLUS of int
type tag_list_cnt = tag list
```

```
(* apply_tag_list_cnt : tag_list_cnt * int -> int *)
let rec apply_tag_list_cnt = function
| ([], a) -> a
| ((SUB2 m) :: cnt, a) -> fib_cps_dfc_tags(m - 2, (PLUS a) :: cnt)
| ((PLUS a) :: cnt, b) -> apply_tag_list_cnt (cnt, a + b)
```

```
(* fib_cps_dfc_tags : (tag_list_cnt * int) -> int *)
and fib_cps_dfc_tags (m, cnt) =
  if m = 0
  then apply_tag_list_cnt(cnt, 1)
  else if m = 1
  then apply_tag_list_cnt(cnt, 1)
  else fib_cps_dfc_tags(m - 1, (SUB2 m) :: cnt)
```

```
(* fib_3 : int -> int *)
let fib_3 m = fib_cps_dfc_tags(m, [])
```

52

Combine Mutually tail-recursive functions into a single function

```

type state_type =
  | SUB1 (* for right-hand-sides starting with fib_ *)
  | APPL (* for right-hand-sides starting with apply_ *)
type state = (state_type * int * tag_list_cnt) -> int

(* eval : state -> int
   A two-state transition function*)
let rec eval = function
  | (SUB1, 0, cnt) -> eval (APPL, 1, cnt)
  | (SUB1, 1, cnt) -> eval (APPL, 1, cnt)
  | (SUB1, m, cnt) -> eval (SUB1, (m-1), (SUB2 m) :: cnt)
  | (APPL, a, (SUB2 m) :: cnt) -> eval (SUB1, (m-2), (PLUS a) :: cnt)
  | (APPL, b, (PLUS a) :: cnt) -> eval (APPL, (a+b), cnt)
  | (APPL, a, []) -> a
  | _ -> failwith "eval : runtime error!"

(* fib_4 : int -> int *)
let fib_4 m = eval (SUB1, m, [])

```

53

The Fibonacci Machine!

```

(* step : state -> state *)
let step = function
  | (SUB1, 0, cnt) -> (APPL, 1, cnt)
  | (SUB1, 1, cnt) -> (APPL, 1, cnt)
  | (SUB1, m, cnt) -> (SUB1, (m-1), (SUB2 m) :: cnt)
  | (APPL, a, (SUB2 m) :: cnt) -> (SUB1, (m-2), (PLUS a) :: cnt)
  | (APPL, b, (PLUS a) :: cnt) -> (APPL, (a+b), cnt)
  | _ -> failwith "step : runtime error!"

(* clearly TAIL RECURSIVE! *)
let rec driver state = function
  | (APPL, a, []) -> a
  | state -> driver (step state)

(* fib_5 : int -> int *)
let fib_5 m = driver (SUB1, m, [])

```

In this version we have simply made the tail-recursive structure very explicit.

54

Here is a trace of fib_5 6.

```

1 SUB1 || 6 || []
2 SUB1 || 5 || [SUB2 6]
3 SUB1 || 4 || [SUB2 6, SUB2 5]
4 SUB1 || 3 || [SUB2 6, SUB2 5, SUB2 4]
5 SUB1 || 2 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3]
6 SUB1 || 1 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3, SUB2 2]
7 APPL || 1 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3, SUB2 2]
8 SUB1 || 0 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3, PLUS 1]
9 APPL || 1 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3, PLUS 1]
10 APPL || 2 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3]
11 SUB1 || 1 || [SUB2 6, SUB2 5, SUB2 4, PLUS 2]
12 APPL || 1 || [SUB2 6, SUB2 5, SUB2 4, PLUS 2]
13 APPL || 3 || [SUB2 6, SUB2 5, SUB2 4]
14 SUB1 || 2 || [SUB2 6, SUB2 5, PLUS 3]
15 SUB1 || 1 || [SUB2 6, SUB2 5, PLUS 3, SUB2 2]
16 APPL || 1 || [SUB2 6, SUB2 5, PLUS 3, SUB2 2]
17 SUB1 || 0 || [SUB2 6, SUB2 5, PLUS 3, PLUS 1]
18 APPL || 1 || [SUB2 6, SUB2 5, PLUS 3, PLUS 1]
19 APPL || 2 || [SUB2 6, SUB2 5, PLUS 3]
20 APPL || 5 || [SUB2 6, SUB2 5]
21 SUB1 || 3 || [SUB2 6, PLUS 5]
22 SUB1 || 2 || [SUB2 6, PLUS 5, SUB2 3]
23 SUB1 || 1 || [SUB2 6, PLUS 5, SUB2 3, SUB2 2]
24 APPL || 1 || [SUB2 6, PLUS 5, SUB2 3, SUB2 2]
25 SUB1 || 0 || [SUB2 6, PLUS 5, SUB2 3, PLUS 1]
26 APPL || 1 || [SUB2 6, PLUS 5, SUB2 3, PLUS 1]
27 APPL || 2 || [SUB2 6, PLUS 5, SUB2 3]
28 SUB1 || 1 || [SUB2 6, PLUS 5, PLUS 2]
29 APPL || 1 || [SUB2 6, PLUS 5, PLUS 2]
30 APPL || 3 || [SUB2 6, PLUS 5]
31 APPL || 8 || [SUB2 6]
32 SUB1 || 4 || [PLUS 8]
33 SUB1 || 3 || [PLUS 8, SUB2 4]
34 SUB1 || 2 || [PLUS 8, SUB2 4, SUB2 3]
35 SUB1 || 1 || [PLUS 8, SUB2 4, SUB2 3, SUB2 2]
36 APPL || 1 || [PLUS 8, SUB2 4, SUB2 3, SUB2 2]
37 SUB1 || 0 || [PLUS 8, SUB2 4, SUB2 3, PLUS 1]
38 APPL || 1 || [PLUS 8, SUB2 4, SUB2 3, PLUS 1]
39 APPL || 2 || [PLUS 8, SUB2 4, SUB2 3]
40 SUB1 || 1 || [PLUS 8, SUB2 4, PLUS 2]
41 APPL || 1 || [PLUS 8, SUB2 4, PLUS 2]
42 APPL || 3 || [PLUS 8, SUB2 4]
43 SUB1 || 2 || [PLUS 8, PLUS 3]
44 SUB1 || 1 || [PLUS 8, PLUS 3, SUB2 2]
45 APPL || 1 || [PLUS 8, PLUS 3, SUB2 2]
46 SUB1 || 0 || [PLUS 8, PLUS 3, PLUS 1]
47 APPL || 1 || [PLUS 8, PLUS 3, PLUS 1]
48 APPL || 2 || [PLUS 8, PLUS 3]
49 APPL || 5 || [PLUS 8]
50 APPL || 13 || []

```

The OCaml file in basic_transformations/fibonacci_machine.ml contains some code for pretty printing such traces....

55

Pause to reflect

- **What have we accomplished?**
- **We have taken a recursive function and turned it into an iterative function that does not require “stack space” for its evaluation (in OCaml)**
- **However, this function now carries with it something akin to its own stack!**
- **We have derived this iterative function in a step-by-step manner where each tiny step is easily proved correct.**
- **Wow!**

56

That was fun! Let's do it again!

```
type expr =  
  | INT of int  
  | PLUS of expr * expr  
  | SUBT of expr * expr  
  | MULT of expr * expr
```

This time we will derive a stack-machine AND a “compiler” that translates expressions into a list of instructions for the machine.

```
(* eval : expr -> int  
  a simple recursive evaluator for expressions *)  
let rec eval = function  
  | INT a      -> a  
  | PLUS(e1, e2) -> (eval e1) + (eval e2)  
  | SUBT(e1, e2) -> (eval e1) - (eval e2)  
  | MULT(e1, e2) -> (eval e1) * (eval e2)
```

57

Here we go again : CPS

```
type cnt_2 = int -> int  
  
type state_2 = expr * cnt_2  
  
(* eval_aux_2 : state_2 -> int *)  
let rec eval_aux_2 (e, cnt) =  
  match e with  
  | INT a      -> cnt a  
  | PLUS(e1, e2) ->  
    eval_aux_2(e1, fun v1 -> eval_aux_2(e2, fun v2 -> cnt(v1 + v2)))  
  | SUBT(e1, e2) ->  
    eval_aux_2(e1, fun v1 -> eval_aux_2(e2, fun v2 -> cnt(v1 - v2)))  
  | MULT(e1, e2) ->  
    eval_aux_2(e1, fun v1 -> eval_aux_2(e2, fun v2 -> cnt(v1 * v2)))  
  
  (* id_cnt : cnt_2 *)  
  let id_cnt (x : int) = x  
  
  (* eval_2 : expr -> int *)  
  let eval_2 e = eval_aux_2(e, id_cnt)
```

58

Defunctionalise!

```
type cnt_3 =
  | ID
  | OUTER_PLUS of expr * cnt_3
  | OUTER_SUBT of expr * cnt_3
  | OUTER_MULT of expr * cnt_3
  | INNER_PLUS of int * cnt_3
  | INNER_SUBT of int * cnt_3
  | INNER_MULT of int * cnt_3

type state_3 = expr * cnt_3

(* apply_3 : cnt_3 * int -> int *)
let rec apply_3 = function
  | (ID, v) -> v
  | (OUTER_PLUS(e2, cnt), v1) -> eval_aux_3(e2, INNER_PLUS(v1, cnt))
  | (OUTER_SUBT(e2, cnt), v1) -> eval_aux_3(e2, INNER_SUBT(v1, cnt))
  | (OUTER_MULT(e2, cnt), v1) -> eval_aux_3(e2, INNER_MULT(v1, cnt))
  | (INNER_PLUS(v1, cnt), v2) -> apply_3(cnt, v1 + v2)
  | (INNER_SUBT(v1, cnt), v2) -> apply_3(cnt, v1 - v2)
  | (INNER_MULT(v1, cnt), v2) -> apply_3(cnt, v1 * v2)
```

59

Defunctionalise!

```
(* eval_aux_2 : state_3 -> int *)
and eval_aux_3 (e, cnt) =
  match e with
  | INT a -> apply_3(cnt, a)
  | PLUS(e1, e2) -> eval_aux_3(e1, OUTER_PLUS(e2, cnt))
  | SUBT(e1, e2) -> eval_aux_3(e1, OUTER_SUBT(e2, cnt))
  | MULT(e1, e2) -> eval_aux_3(e1, OUTER_MULT(e2, cnt))

(* eval_3 : expr -> int *)
let eval_3 e = eval_aux_3(e, ID)
```

60

Eureka! Again we have a stack!

```
type tag =
| O_PLUS of expr
| I_PLUS of int
| O_SUBT of expr
| I_SUBT of int
| O_MULT of expr
| I_MULT of int

type cnt_4 = tag list
type state_4 = expr * cnt_4

(* apply_4 : cnt_4 * int -> int *)
let rec apply_4 = function
| ([], v) -> v
| ((O_PLUS e2) :: cnt, v1) -> eval_aux_4(e2, (I_PLUS v1) :: cnt)
| ((O_SUBT e2) :: cnt, v1) -> eval_aux_4(e2, (I_SUBT v1) :: cnt)
| ((O_MULT e2) :: cnt, v1) -> eval_aux_4(e2, (I_MULT v1) :: cnt)
| ((I_PLUS v1) :: cnt, v2) -> apply_4(cnt, v1 + v2)
| ((I_SUBT v1) :: cnt, v2) -> apply_4(cnt, v1 - v2)
| ((I_MULT v1) :: cnt, v2) -> apply_4(cnt, v1 * v2)
```

61

Eureka! Again we have a stack!

```
(* eval_aux_4 : state_4 -> int *)
and eval_aux_4 (e, cnt) =
  match e with
  | INT a -> apply_4(cnt, a)
  | PLUS(e1, e2) -> eval_aux_4(e1, O_PLUS(e2) :: cnt)
  | SUBT(e1, e2) -> eval_aux_4(e1, O_SUBT(e2) :: cnt)
  | MULT(e1, e2) -> eval_aux_4(e1, O_MULT(e2) :: cnt)

(* eval_4 : expr -> int *)
let eval_4 e = eval_aux_4(e, [])
```

62

Eureka! Can combine apply_4 and eval_aux_4

```
type acc =  
  | A_INT of int  
  | A_EXP of expr  
  
type cnt_5 = cnt_4  
  
type state_5 = cnt_5 * acc  
  
val : step : state_5 -> state_5  
  
val driver : state_5 -> int  
  
val eval_5 : expr -> int
```

Type of an “accumulator” that contains either an int or an expression.

The driver will be clearly tail-recursive ...

63

Rewrite to use driver, accumulator

```
let step_5 = function  
  | (cnt, A_EXP (INT a)) -> (cnt, A_INT a)  
  | (cnt, A_EXP (PLUS(e1, e2))) -> (O_PLUS(e2) :: cnt, A_EXP e1)  
  | (cnt, A_EXP (SUBT(e1, e2))) -> (O_SUBT(e2) :: cnt, A_EXP e1)  
  | (cnt, A_EXP (MULT(e1, e2))) -> (O_MULT(e2) :: cnt, A_EXP e1)  
  | ((O_PLUS e2) :: cnt, A_INT v1) -> ((I_PLUS v1) :: cnt, A_EXP e2)  
  | ((O_SUBT e2) :: cnt, A_INT v1) -> ((I_SUBT v1) :: cnt, A_EXP e2)  
  | ((O_MULT e2) :: cnt, A_INT v1) -> ((I_MULT v1) :: cnt, A_EXP e2)  
  | ((I_PLUS v1) :: cnt, A_INT v2) -> (cnt, A_INT (v1 + v2))  
  | ((I_SUBT v1) :: cnt, A_INT v2) -> (cnt, A_INT (v1 - v2))  
  | ((I_MULT v1) :: cnt, A_INT v2) -> (cnt, A_INT (v1 * v2))  
  | ([], A_INT v) -> ([], A_INT v)  
  
let rec driver_5 = function  
  | ([], A_INT v) -> v  
  | state -> driver_5 (step_5 state)  
  
let eval_5 e = driver_5([], A_EXP e)
```

64

Eureka! There are really two independent stacks here --- one for “expressions” and one for values

```
type directive =
| E of expr
| DO_PLUS
| DO_SUBT
| DO_MULT

type directive_stack = directive list

type value_stack = int list

type state_6 = directive_stack * value_stack

val step_6 : state_6 -> state_6

val driver_6 : state_6 -> int

val eval_6 : expr -> int
```

The state is now two stacks!

65

Split into two stacks

```
let step_6 = function
| (E(INT v) :: ds, vs) -> (ds, v :: vs)
| (E(PLUS(e1, e2)) :: ds, vs) -> ((E e1) :: (E e2) :: DO_PLUS :: ds, vs)
| (E(SUBT(e1, e2)) :: ds, vs) -> ((E e1) :: (E e2) :: DO_SUBT :: ds, vs)
| (E(MULT(e1, e2)) :: ds, vs) -> ((E e1) :: (E e2) :: DO_MULT :: ds, vs)
| (DO_PLUS :: ds, v2 :: v1 :: vs) -> (ds, (v1 + v2) :: vs)
| (DO_SUBT :: ds, v2 :: v1 :: vs) -> (ds, (v1 - v2) :: vs)
| (DO_MULT :: ds, v2 :: v1 :: vs) -> (ds, (v1 * v2) :: vs)
| _ -> failwith "eval : runtime error!"

let rec driver_6 = function
| ([], [v]) -> v
| state -> driver_6 (step_6 state)

let eval_6 e = driver_6 ([E e], [])
```

66

Look closely

This evaluator is interleaving two distinct computations:

- (1) decomposition of the input expression into sub-expressions
- (2) the computation of +, -, and *.

Idea: why not do the decomposition BEFORE the computation?

67

Refactor --- compile!

```
type instr =  
  | Ipush of int  
  | Iplus  
  | Isubt  
  | Imult  
  
type code = instr list  
  
type state_? = code * value_stack  
  
(* compile : expr -> code *)  
let rec compile = function  
  | INT a      -> [Ipush a]  
  | PLUS(e1, e2) -> (compile e1) @ (compile e2) @ [Iplus]  
  | SUBT(e1, e2) -> (compile e1) @ (compile e2) @ [Isubt]  
  | MULT(e1, e2) -> (compile e1) @ (compile e2) @ [Imult]
```

68

Evaluate compiled code.

```
(* step_7 : state_7 -> state_7 *)
let step_7 = function
| (Ipush v :: is,      vs) -> (is, v :: vs)
| (Iplus :: is, v2::v1::vs) -> (is, (v1 + v2) :: vs)
| (Isubt :: is, v2::v1::vs) -> (is, (v1 - v2) :: vs)
| (Imult :: is, v2::v1::vs) -> (is, (v1 * v2) :: vs)
| _ -> failwith "eval : runtime error!"

let rec driver_7 = function
| ([], [v]) -> v
| _ -> driver_7 (step_7 state)

let eval_7 e = driver_7 (compile e, []) 1
```

69

A trace

```
compile (PLUS(INT 89, MULT(INT 2, SUBT(INT 10, INT 4))))
= [add; mul; sub; push 4; push 10; push 2; push 89]
```

pretty
printed!

```
state 1 IS = [add; mul; sub; push 4; push 10; push 2; push 89]
        VS = []
state 2 IS = [add; mul; sub; push 4; push 10; push 2]
        VS = [89]
state 3 IS = [add; mul; sub; push 4; push 10]
        VS = [89; 2]
state 4 IS = [add; mul; sub; push 4]
        VS = [89; 2; 10]
state 5 IS = [add; mul; sub]
        VS = [89; 2; 10; 4]
state 6 IS = [add; mul]
        VS = [89; 2; 6]
state 7 IS = [add]
        VS = [89; 12]
state 8 IS = []
        VS = [101]
```

Top of each
stack is on
the right

70

Pause to reflect

- What have we accomplished?
- We have taken a recursive function and turned it into a iterative function that does not require “stack space” for its evaluation (in OCaml)
- However, this function now carries with it something akin to its own stack!
- We have derived this iterative function in a step-by-step manner where each tiny step is easily proved correct.
- **This time we have gone one step further than with the Fibonacci Machine --- we have refactored the evaluation into two steps. 1) compilation and 2) evaluation of compiled code.**

It is not so apparent with our expression evaluator --- since we are not taking any “input” from the external world --- but this highlights one difference between an interpreter and a Virtual Machine. When using a VM, the compiler does a lot of analysis and rewriting once upfront, leaving the code for multiple executions.

71