Why Study Compilers?

- Although many of the basic ideas were developed over 60 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.
Compilation is a special kind of translation

Source Program Text → The compiler → program for target “machine”

Just text – no way to run program! → We have a “machine” to run this!

A good compiler should ...
- be correct in the sense that meaning is preserved
- produce usable error messages
- generate efficient code
- itself be efficient
- be well-structured and maintainable

This course! OptComp, Part II

Pick any 2? Just 1?

Mind The Gap

<table>
<thead>
<tr>
<th>High Level Language</th>
<th>Typical Target Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Machine” independent</td>
<td>“Machine” specific</td>
</tr>
<tr>
<td>Complex syntax</td>
<td>Simple syntax</td>
</tr>
<tr>
<td>Complex type system</td>
<td>Simple types</td>
</tr>
<tr>
<td>Variables</td>
<td>memory, registers, words</td>
</tr>
<tr>
<td>Nested scope</td>
<td>Single flat scope</td>
</tr>
<tr>
<td>Procedures, functions</td>
<td></td>
</tr>
<tr>
<td>Objects</td>
<td></td>
</tr>
<tr>
<td>Modules</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
</tbody>
</table>

Help!!! Where do we begin???
The Gap, illustrated

public class Fibonacci {
    public Fibonacci() {
        Code:
        0: aload_0
        1: invokespecial #1
        4: return
    }
    public static long fib(int m) {
        Code:
        0: iload_0
        1: ifne          6
        4: lconst_1
        5: lreturn
        6: iload_0
        7: icomparene 13
        11: iload_1
        12: return
        13: iload_0
        14: iload_1
        15: isub
        16: invokespecial #2
        19: iload_0
        20: iload_2
        21: isub
        22: invokespecial #2
        25: ladd
        26: lreturn
    }
    public static void main(String[] args) {
        int m =
        Integer.parseInt(args[0]);
        System.out.println(
        "fib(m) + \"n\";
    }
}

class Fibonacci {
    private 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 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

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

The Gap, illustrated

branch L2
L1: acc 0
    push
    const 0
    eqint
    branchifnot L4
    const 1
    return
L4: acc 0
    push
    const 1
    eqint
    branchifnot L3
    const 1
    return
L3: acc 0
    offsetint -2
    push
    offsetclosure 0
    apply
    push
    acc 1
    offsetint -1
    push
    offsetclosure 0
    apply
    addint
    return
L2: closurerec 1, 0
    acc 0
    makeblock 1, 0
    pop
    setglobal Fib!

OCaml VM bytecodes
The Gap, illustrated

fib.c

#include<stdio.h>

int Fibonacci(int);
int main()
{
    int n;
    scanf("%d", &n);
    printf("%d
", Fibonacci(n));
    return 0;
}

int Fibonacci(int n)
{
    if ( n == 0 ) return 0;
    else if ( n == 1 ) return 1;
    else return ( Fibonacci(n - 1) + Fibonacci(n - 2) );
}

gcc -S fib.c

The Gap, illustrated
Key to bridging The Gap: divide and conquer. The Big Leap is broken into small steps. Each step broken into yet smaller steps …
The middle

- High-level to low-level
- Optimisations

Low-level retargetable representation

Trade-off: with more optimisations the generated code is (normally) faster, but the compiler is slower

The back-end

- JVM bytecodes
- x86/Linux
- x86/MacOS
- x86/FreeBSD
- x86/Windows
- ARM/Android
- ...
- ...

- Requires intimate knowledge of instruction set and details of target machine
- When generating assembler, need to understand details of OS interface
- Target-dependent optimisations happen here!
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?

The Slang compiler

- The compiler is available from the course web site.
- It is written in Ocaml
- Slang = Simple Language. Based on L3 from Semantics of Programming Languages, Part 1B.
- The best way to learn about compilers is to modify one.
- There are several suggested improvements listed on the course web site. I hope that some of you will implement these. If they work, I’ll let you commit your changes to the repository. Fame! Fortune!
**Question**: How do we leap from the mathematical semantics of L3 to a low-level stack machine?

**Answer**: We will start with a high-level interpreter based on semantics, and then derive the stack machine by a sequence of semantics preserving transformations!

**Lectures 2 – 6: the derivation**

Note: this is not the traditional way of teaching compilers! Many textbooks will start with a stack machine and bridge the gap informally. We will develop a deeper understanding!
The Shape of this Course

• Lectures 2 – 6: Derivation of Jargon Stack Machine
• Lectures 7 – 12: Assorted topics.
  • From Jargon code to x86
  • The OS interface
  • Implementing exceptions
  • Implementing OOP
  • Runtime systems and garbage collection
  • Compiling a compiler via boot strapping
• Lectures 13 – 16: The front end.
  • Lexical analysis
  • Parsing

LECTURE 2
Slang front end and interpreter 0

• Slang (= Simple LANGUAGE)
  – A subset of L3 from Semantics …
  – … with very ugly concrete syntax
  – You are invited to experiment with improvements to this concrete syntax.
• Slang: concrete syntax, types
• Abstract Syntax Trees (ASTs)
• The Front End
• Interpreter 0: The high-level “definitional” interpreter
  1. Slang/L3 values represented directly as OCaml values
  2. Recursive interpreter implements a denotational semantics
  3. The interpreter implicitly uses OCaml’s runtime stack and heap
Clunky Slang Syntax (informal)

uop := - | ~

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

t ::= bool | int | unit | (t) | t * t | t + t | t -> t | t ref

e ::= () | n | true | false | x | (e) | ? |

e bop e | uop e |
if e then else e end |
e e | fun (x : t) -> e end |
let x : t = e in e end |
let f(x : t) : t = e in e end |
!e | ref e | e := e | while e do e end |
begin e; e; ... e end |
(e, e) | snd e | fst e |
inl t e | inr t e |
case e of inl(x : t) -> e | inr(x : t) -> e end

(~ is boolean negation)

(? requests an integer input from terminal)

(notice type annotation on inl and inr constructs)

From slang/examples

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

let gcd( p : int * int) : int = 
let m : int = fst p
in let n : int = snd p
in if m = n
then m
else if m < n
then gcd(m, n - m)
else gcd(m - n, n)
end
end
end
in gcd(? , ?) end

The ? requests an integer input from the terminal
### Slang Front End

**Input file foo.slang**

- **Parse (we use Ocaml versions of LEX and YACC, covered in Lectures 3 --- 6)**
- **Parsed AST (Past.expr)**
- **Static analysis: check types, and context-sensitive rules, resolve overloaded operators**
- **Parsed AST (Past.expr)**
- **Remove “syntactic sugar”, file location information, and most type information**
- **Intermediate AST (Ast.expr)**

---

#### Parsed AST (past.ml)

**Type Definitions**

- `type var = string`
- `type loc = Lexing.position`
- `type type_expr =`
  - `TEint`
  - `TEbool`
  - `TEunit`
  - `TEref of type_expr`
  - `TEarrow of type_expr * type_expr`
  - `TEproduct of type_expr * type_expr`
  - `TEunion of type_expr * type_expr`
- `type oper = ADD | MUL | SUB | LT | AND | OR | EQ | EQB | EQI`
- `type unary_oper = NEG | NOT`

**Locations (loc) are used in generating error messages.**
val infer : (Past.var * Past.type_expr) list
  -> (Past.expr * Past.type_expr)

val check : Past.expr -> Past.expr (* infer on empty environment *)

- Check type correctness
- Rewrite expressions to resolve EQ to EQI (for integers)
  or EQB (for bools).
- Only LetFun is returned by parser. Rewrite to LetRecFun
  when function is actually recursive.

Lesson: while enforcing “context-sensitive rules” we can resolve
ambiguities that cannot be specified in context-free grammars.

---

Internal AST (ast.ml)

| type var = string |
| type oper = ADD | MUL | SUB | LT | AND | OR | E QB | EQ I |
| type unary_oper = NEG | NOT | READ |

No locations, types. No Let, EQ.

Is getting rid of types a bad idea? Perhaps a full answer would be
language-dependent…

| type expr = |
| Unit |
| Var of var |
| Integer of int |
| Boolean of bool |
| UnaryOp of unary_oper * expr |
| Op of expr * oper * expr |
| If of expr * expr * expr |
| Pair of expr * expr |
| Fst of expr |
| Snd of expr |
| Inl of expr |
| Inr of expr |
| Case of expr * lambda * lambda |
| While of expr * expr |
| Seq of (expr list) |
| Ref of expr |
| Der ef of expr |
| Assign of expr * expr |
| Lambda of lambda |
| App of expr * expr |
| LetFun of var * lambda * expr |
| LetRecFun of var * lambda * expr |

and lambda = var * expr
This is done to simplify some of our code. Is it a good idea? Perhaps not!

Approaches to Mathematical Semantics

- **Axiomatic**: Meaning defined through logical specifications of behaviour.
  - Hoare Logic (Part II)
  - Separation Logic
- **Operational**: Meaning defined in terms of transition relations on states in an abstract machine.
  - Semantics (Part 1B)
- **Denotational**: Meaning is defined in terms of mathematical objects such as functions.
  - Denotational Semantics (Part II)
A denotational semantics for L3?

\[ \begin{align*}
A &= \text{set of addresses} \\
B &= \text{set of booleans} \\
I &= \text{set of identifiers} \\
Expr &= \text{set of L3 expressions} \\
E &= \text{set of environments} = I \rightarrow V \\
S &= \text{set of stores} = A \rightarrow V \\
V &= \text{set of value} \\
&= A + N + B + \{ () \} + V \times V + (V + V) + (V \times S) \rightarrow (V \times S)
\end{align*} \]

\[ \begin{align*}
N &= \text{set of integers} \\
B &= \text{set of booleans} \\
I &= \text{set of identifiers} \\
E &= \text{set of environments} = I \rightarrow V \\
V &= \text{set of value} \\
&= A + N + B + \{ () \} + V \times V + (V + V) + (V \times S) \rightarrow (V \times S)
\end{align*} \]

\[ M = \text{the meaning function} \]
\[ M : (Expr \times E \times S) \rightarrow (V \times S) \]

Set of values \( V \) solves this “domain equation” (here + means disjoint union).

Solving such equations is where some difficult maths is required …

Interpreter 0: An OCaml approximation

\[ \begin{align*}
A &= \text{set of addresses} \\
S &= \text{set of stores} = A \rightarrow V \\
V &= \text{set of value} \\
&= A + N + B + \{ () \} + V \times V + (V + V) + (V \times S) \rightarrow (V \times S)
\end{align*} \]

\[ \begin{align*}
E &= \text{set of environments} = A \rightarrow V \\
M &= \text{the meaning function} \\
M : (Expr \times E \times S) \rightarrow (V \times S)
\end{align*} \]

\[ \begin{align*}
\text{type} \ address \\
\text{type} \ store = address \rightarrow value \\
\text{and} \ value =
&\text{REF of address} \\
&\text{INT of int} \\
&\text{BOOL of bool} \\
&\text{UNIT} \\
&\text{PAIR of value * value} \\
&\text{INL of value} \\
&\text{INR of value} \\
&\text{FUN of ((value * store) \rightarrow (value * store))}
\end{align*} \]

\[ \begin{align*}
\text{type} \ env = \text{Ast.var} \rightarrow value \\
\text{val} \ interpret : \text{Ast.expr} * \text{env} * \text{store} \rightarrow (value * store)
\end{align*} \]
Most of the code is obvious!

```ocaml
let rec interpret (e, env, store) =
  match e with
  | If(e1, e2, e3) ->
    let (v, store') = interpret(e1, env, store) in
    (match v with
      | BOOL true -> interpret(e2, env, store')
      | BOOL false -> interpret(e3, env, store')
      | _ -> complain "runtime error. Expecting a boolean!")
  | Pair(e1, e2) ->
    let (v1, store1) = interpret(e1, env, store) in
    let (v2, store2) = interpret(e2, env, store1) in
    (PAIR v1, v2), store2
  | Fst e ->
    (match interpret(e, env, store) with
      | (PAIR _, v2), store' -> (v2, store')
      | _ -> complain "runtime error. Expecting a pair!")
  | Snd e ->
    (match interpret(e, env, store) with
      | (PAIR v1, _) -> (v1, store')
      | |_ -> complain "runtime error. Expecting a pair!")
  | Inl e ->
    let (v, store') = interpret(e, env, store) in
    (INL v, store')
  | Inr e ->
    let (v, store') = interpret(e, env, store) in
    (INR v, store')
```

Tricky bits: Slang functions mapped to OCaml functions!

```ocaml
let rec interpret (e, env, store) =
  match e with
  | Lambda(x, e) -> (FUN (fun (v, s) -> interpret(e, update(env, (x, v)), s)), store)
  | App(e1, e2) -> (* I chose to evaluate argument first! *)
    let (v2, store1) = interpret(e2, env, store) in
    let (v1, store2) = interpret(e1, env, store1) in
    (match v1 with
      | FUN f -> f (v2, store2)
      | _ -> complain "runtime error. Expecting a function!")
  | LetFun(f, (x, body), e) ->
    let new_env =
      update(env, (f, FUN (fun (v, s) -> interpret(body, update(env, (x, v)), s)))))
    in
    interpret(e, new_env, store)
  | LetRecFun(f, (x, body), e) ->
    let rec new_env g = (* a recursive environment!!! *)
      if g = f then FUN (fun (v, s) -> interpret(body, update(new_env, (x, v)), s))
      else env g
    in
    interpret(e, new_env, store)

update : env * (var * value) -> env
```
Interpreter 0 is using OCaml's runtime stack. How can we move toward the Jargon VM?

The run-time data structure is the call stack containing an activation record for each function invocation.

```ocaml
let fun f (x) = x + 1
fun g(y) = f(y+2)+2
fun h(w) = g(w+1)+3
in
    h(h(17))
end
```

Recall tail recursion: fold_left vs fold_right

From ocaml-4.01.0/stdlib/list.ml:

```ocaml
(* fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
   fold_left f a [b1; ...; bn] = 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
Convert tail-recursion to iteration

(* 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)

(* 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 rm := !rm - !rn
      else rm := !rn - !rm
    done
  in !result

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.

Question: can we transform any recursive function (such as interpreter 0) 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!

We will apply this kind of transformation to the code of interpreter 0 as the first steps towards deriving interpreter 1.
LECTURES 3 & 4
Derivation of Interpreters 1 & 2

• 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”!
• This provides a roadmap for the interp_0 → interp_1 → interp_2 derivations.

(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)))
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.

```ocaml
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)))
```

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

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

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

Expressed with “let” rather than “fun”

```ocaml
(* 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  
....
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]

Correctness?

For all c : int -> int, for all m, 0 <= m, we have, c(fib m) = fib_cps(m, c).

Proof: assume c : int -> int. By Induction on m. Base case : m = 0:
  fib_cps(0, c) = c(1) = c(fib(0).

Induction step: Assume for all n < m, c(fib n) = 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)

NB: This proof pretends that we can treat OCaml functions as ideal mathematical functions, which of course we cannot. OCaml functions might raise exceptions like "stack overflow" or "you burned my toast", and so on. But this is a convenient fiction as long as we remember to be careful.
Correctness?

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

Can with express fib_cps without a functional argument?

(* 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
* Datatype to represent continuations *

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

* apply_cnt : cnt * int -> int *

```plaintext
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 *

```plaintext
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 *

```plaintext
let fib_2 m = fib_cps_dfc(m, ID)
```

---

**Correctness?**

Let `< c >` be of type cnt representing a continuation `c : int -> int` constructed by `fib_cps`.

Then

- `apply_cnt(< c >, m) = c(m)`
- `fib_cps(n, c) = fib_cps_dfc(n, < c >)`.

---

Functional continuation `c` Representation `< c >`

| `fun a -> fib_cps(m - 2, fun b -> cnt (a + b))` | `CNT1(m, < cnt >)` |
| `fun b -> cnt (a + b)` | `CNT2(a, < cnt >)` |
| `fun x -> x` | `ID` |
Eureka! Continuations are just lists
(used like a stack)

\[\text{type } \text{int\_list} = \text{NIL} \mid \text{CONS of int} \times \text{int\_list}\]

\[\text{type } \text{cnt} = \text{ID} \mid \text{CNT1 of int} \times \text{cnt} \mid \text{CNT2 of int} \times \text{cnt}\]

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

\[\text{type } \text{tag} = \text{SUB2 of int} \mid \text{PLUS of int}\]

\[\text{type } \text{tag\_list\_cnt} = \text{tag list}\]

The continuation lists are used like a stack!

\[\text{type } \text{tag} = \text{SUB2 of int} \mid \text{PLUS of int}\]

\[\text{type } \text{tag\_list\_cnt} = \text{tag list}\]

\[\text{(* apply\_tag\_list\_cnt : tag\_list\_cnt} \times \text{int} \rightarrow \text{int} *)\]

\[\text{let rec apply\_tag\_list\_cnt = function}\]
\[\begin{align*}
&| ([], a) \rightarrow a \\
&| ((\text{SUB2} m) :: \text{cnt}, a) \rightarrow \text{fib\_cps\_dfc\_tags}(m - 2, (\text{PLUS} a) :: \text{cnt}) \\
&| ((\text{PLUS} a) :: \text{cnt}, b) \rightarrow \text{apply\_tag\_list\_cnt}(\text{cnt}, a + b)
\end{align*}\]

\[\text{(* fib\_cps\_dfc\_tags : (tag\_list\_cnt} \times \text{int}) \rightarrow \text{int} *)\]

\[\text{and fib\_cps\_dfc\_tags (m, cnt) =}\]
\[\begin{align*}
&\text{if } m = 0 \\
&\text{then apply\_tag\_list\_cnt}(\text{cnt}, 1) \\
&\text{else if } m = 1 \\
&\text{then apply\_tag\_list\_cnt}(\text{cnt}, 1) \\
&\text{else fib\_cps\_dfc\_tags}(m - 1, (\text{SUB2} m) :: \text{cnt})
\end{align*}\]

\[\text{(* fib\_3 : int} \rightarrow \text{int} *)\]

\[\text{let fib\_3 m = fib\_cps\_dfc\_tags(m, [])}\]
Combine Mutually tail-recursive functions into a single function

```ocaml
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

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

Eliminate tail recursion to obtain The Fibonacci Machine!

```ocaml
(* 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.
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, PLUS 1]
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, PLUS 2]
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, PLUS 1]
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....

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 its own evaluation stack as an extra argument!
• We have derived this iterative function in a step-by-step manner where each tiny step is easily proved correct.
• Wow!
That was fun! Let’s do it again!

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)

Here we go again: CPS

(* eval : expr -> int
a simple recursive evaluator for expressions *)

type cnt_2 = int -> int

(* eval_aux : state -> int *)

let rec eval_aux (e, cnt) = match e with
| INT a -> cnt a
| PLUS(e1, e2) ->
  eval_aux(e1, fun v1 ->
    eval_aux(e2, fun v2 ->
      cnt(v1 + v2)))
| SUBT(e1, e2) ->
  eval_aux(e1, fun v1 ->
    eval_aux(e2, fun v2 ->
      cnt(v1 - v2)))
| MULT(e1, e2) ->
  eval_aux(e1, fun v1 ->
    eval_aux(e2, fun v2 ->
      cnt(v1 * v2)))

(* id_cnt : cnt *)

let id_cnt (x : int) = x

(* eval_2 : expr -> int *)

let eval_2 e = eval_aux(e, id_cnt)
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)
```

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)
```
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)

(* 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, [])

Eureka! Again we have a stack!
Eureka! Can combine apply_4 and eval_aux_4

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

The driver will be clearly tail-recursive …

---

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)
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 exp_6 : expr -> int
```

The state is now two stacks!

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], [])
```
An eval_6 trace

\[ e = \text{PLUS}\left(\text{MULT}(\text{INT} 89, \text{INT} 2), \text{SUBT}(\text{INT} 10, \text{INT} 4)\right) \]

Top of each stack is on the right

<table>
<thead>
<tr>
<th>State</th>
<th>DS</th>
<th>VS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[E(PLUS(MULT(INT(89), INT(2)), SUBT(INT(10), INT(4))))]</td>
<td>[]</td>
</tr>
<tr>
<td>2</td>
<td>[DO_PLUS; E(SUBT(INT(10), INT(4))); E(MULT(INT(89), INT(2)))]</td>
<td>[]</td>
</tr>
<tr>
<td>3</td>
<td>[DO_PLUS; E(SUBT(INT(10), INT(4))); DO_MULT; E(INT(2)); E(INT(89))]</td>
<td>[]</td>
</tr>
<tr>
<td>4</td>
<td>[DO_PLUS; E(SUBT(INT(10), INT(4))); DO_MULT; E(INT(2))]</td>
<td>[89]</td>
</tr>
<tr>
<td>5</td>
<td>[DO_PLUS; E(SUBT(INT(10), INT(4))); DO_MULT]</td>
<td>[89; 2]</td>
</tr>
<tr>
<td>6</td>
<td>[DO_PLUS; E(SUBT(INT(10), INT(4)))]</td>
<td>[89; 2]</td>
</tr>
<tr>
<td>7</td>
<td>[DO_PLUS; DO_SUBT; E(INT(4)); E(INT(10))]</td>
<td>[178]</td>
</tr>
<tr>
<td>8</td>
<td>[DO_PLUS; DO_SUBT; E(INT(4))]</td>
<td>[178; 10]</td>
</tr>
<tr>
<td>9</td>
<td>[DO_PLUS; DO_SUBT]</td>
<td>[178; 10; 4]</td>
</tr>
<tr>
<td>10</td>
<td>[DO_PLUS]</td>
<td>[178; 6]</td>
</tr>
<tr>
<td>11</td>
<td>[]</td>
<td>[184]</td>
</tr>
</tbody>
</table>

Key insight

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?

Key insight: An interpreter can (usually) be **refactored** into a translation (compilation!) followed by a lower-level interpreter.

\[ \text{Interpret}_\text{higher} \left( e \right) = \text{interpret}_\text{lower}(\text{compile}(e)) \]

Note: This can occur at many levels of abstraction: think of machine code being interpreted in micro-code …
Refactor --- compile!

(* low-level instructions *)

type instr =
| Ipush of int
| Iplus
| Isubt
| Imult

type code = instr list

type state_7 = 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]

Never put off till run-time what you can do at compile-time.

---

David Gries

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, [])
An eval_7 trace

```
compile (PLUS(MULT(INT 89, INT 2), SUBT(INT 10, INT 4)))
= [push 89; push 2; mult; push 10; push 4; subt; plus]
```

state 1   IS = [add; sub; push 4; push 10; mul; push 2; push 89]
VS = []

state 2   IS = [add; sub; push 4; push 10; mul; push 2]
VS = [89]

state 3   IS = [add; sub; push 4; push 10; mul]
VS = [89; 2]

state 4   IS = [add; sub; push 4; push 10]
VS = [178]

state 5   IS = [add; sub; push 4]
VS = [178; 10]

state 6   IS = [add; sub]
VS = [178; 10; 4]

state 7   IS = [add]
VS = [178; 6]

state 8   IS = []
VS = [184]
```

interpret is implicitly using Ocaml’s runtime stack

```
let rec interpret (e, env, store) =
  match e with
  | Integer n -> (INT n, store)
  | Op(e1, op, e2) ->
    let (v1, store1) = interpret(e1, env, store) in
    let (v2, store2) = interpret(e2, env, store1) in
    (do_oper(op, v1, v2), store2)
  | ... |
```

- Every invocation of interpret is building an activation record on Ocaml’s runtime stack.
- **We will now define interpreter 2 which makes this stack explicit**
The derivation from eval to compile+eval_7 can be used as a guide to a derivation from Interpreter 0 to interpreter 2.

1. Apply CPS to the code of Interpreter 0
2. Defunctionalise
3. Arrive at interpreter 1, which has a single continuation stack containing expressions, values and environments (analogous to eval_6)
4. Split this stack into two stacks: one for instructions and the other for values and environments
5. Refactor into compiler + lower-level interpreter
6. Arrive at interpreter 2. (analogous to eval_7)

Interpreter 2: A high-level stack-oriented machine

1. Makes the Ocaml runtime stack explicit
2. Complex values pushed onto stacks
3. One stack for values and environments
4. One stack for instructions
5. Heap used only for references
6. Instructions have tree-like structure

(we will not look at the details of interpreter 1 …)
In_interp_2 data types

type address

type store = address -> value

and value =
  | REF of address
  | INT of int
  | BOOL of bool
  | UNIT
  | PAIR of value * value
  | INL of value
  | INR of value
  | FUN of ((value * store) -> (value * store))

type env = Ast.var -> value

and closure = code * env

and code = instruction list

val step : state -> state
val driver : state -> value
val compile : expr -> code
val interpret : expr -> value

Interp_0

Interp_2

Interp_2.ml : The Abstract Machine

The state is actually comprised of a heap --- a global array of values --- a pair of the form

(code, evn_value_stack)
Interpreter 2: The Abstract Machine

The state transition function.

let step = function
  (value/env stack) -> (code stack, value/env stack)

val step : state -> state

The driver. Correctness

(* val driver : state -> value *)

let rec driver state =
  match state with
  | ([], [V v]) -> v
  | _ -> driver (step state)

val compile : expr -> code

The idea: if e passes the frond-end and
Interp_0.interpret e = v
then
driver (compile e, []) = v'
where v' (somehow) represents v.
Implement inter_0 in interp_2

```ocaml
let rec interpret (e, env, store) = match e with
  | Pair(e1, e2) ->
    let (v1, store1) = interpret(e1, env, store) in
    let (v2, store2) = interpret(e2, env, store1) in (PAIR(v1, v2), store2)
  | Fst e ->
    (match interpret(e, env, store) with
     | (PAIR (v1, _), store') -> (v1, store')
     | (v, _) -> complain "runtime error. Expecting a pair!")

let step = function
  | (MK_PAIR :: ds, (V v2) :: (V v1) :: evs) -> (ds, V(PAIR(v1, v2)) :: evs)
  | (FST :: ds, V(PAIR (v, _)) :: evs) -> (ds, (V v) :: evs)

let rec compile = function
  | Pair(e1, e2) -> (compile e1) @ (compile e2) @ [MK_PAIR]
  | Fst e -> (compile e) @ [FST]

let rec interpret (e, env, store) = match e with
  | If(e1, e2, e3) ->
    let (v, store') = interpret(e1, env, store) in
    (match v with
     | BOOL true -> interpret(e2, env, store')
     | BOOL false -> interpret(e3, env, store')
     | v -> complain "runtime error. Expecting a boolean!")
```

interp_0.ml

Interpretation of pair expressions and conditional expressions in the interpreter module.

Implement inter_0 in interp_2

```ocaml
let rec interpret (e, env, store) =
  match e with
  | If(e1, e2, e3) ->
    let (v, store') = interpret(e1, env, store) in
    (match v with
     | BOOL true -> interpret(e2, env, store')
     | BOOL false -> interpret(e3, env, store')
     | v -> complain "runtime error. Expecting a boolean!")

let step = function
  | ((TEST(c1, c2)) :: ds, V(BOOL true) :: evs) -> (c1 @ ds, evs)
  | ((TEST(c1, c2)) :: ds, V(BOOL false) :: evs) -> (c2 @ ds, evs)

let rec compile = function
  | If(e1, e2, e3) -> (compile e1) @ [TEST(compile e2, compile e3)]
```

interp_2.ml

Conditional expressions are interpreted based on the truth value of the condition.
Tricky bits again!

```
let rec interpret (e, env, store) =
  match e with
  | Lambda(x, e)  -> (FUN (fun (v, s) -> interpret(e, update(env, (x, v)), s)), store)
  | App(e1, e2)   -> (* I chose to evaluate argument first! *)
    let (v2, store1) = interpret(e2, env, store) in
    let (v1, store2) = interpret(e1, env, store1) in
    (match v1 with
      | FUN f -> f (v2, store2)
      | v    -> complain "runtime error. Expecting a function!")
  :
```

```
let step = function
  | (POP :: ds, s :: evs)  -> (ds, evs)
  | (SWAP :: ds, s1 :: s2 :: evs)  -> (ds, s2 :: s1 :: evs)
  | ((BIND x) :: ds, (V v) :: evs) -> (ds, EVI([(x, v)]) :: evs)
  | ((MK_CLOSURE c) :: ds, evs)   -> (ds, V(mk_fun(c, evs_to_env evs)) :: evs)
  | (APPLY :: ds, V(CLOSURE (_, (c, env))) :: (V v) :: evs)
    -> (c @ ds, (V v) :: (EV env) :: evs)
  :
```

```
let rec compile = function
  | Lambda(x, e)  -> [MK_CLOSURE((BIND x) :: (compile e) @ [SWAP; POP])]
  | App(e1, e2)   -> (compile e2) @ (compile e1) @ [APPLY; SWAP; POP]
  :
```

Example: Compiled code for rev_pair.slang

```
let rev_pair (p : int * int) : int * int = (snd p, fst p)
in
  rev_pair (21, 17)
end

MK_CLOSURE((BIND p; LOOKUP p; SND; LOOKUP p; FST; MK_PAIR; SWAP; POP));
BIND rev_pair;
PUSH 21;
PUSH 17;
MK_PAIR;
LOOKUP rev_pair;
APPLY;
SWAP;
POP;
SWAP;
POP

DEMO TIME!!!
```
LECTURE 5
Derive Interpreter 3

1. “Flatten” code into linear array
2. Add “code pointer” (cp) to machine state
3. New instructions : LABEL, GOTO, RETURN
4. “Compile away” conditionals and while loops

Linearise code

Interpreter 2 copies code on the code stack.
We want to introduce one global array of instructions indexed by a code pointer (cp).
At runtime the cp points at the next instruction to be executed.

This will require two new instructions:

LABEL L : Associate label L with this location in the code array
GOTO L : Set the cp to the code address associated with L
Compile conditionals, loops

**If** \((e_1, e_2, e_3)\)

- code for \(e_1\)
- TEST \(k\)
- code for \(e_2\)
- GOTO \(m\)
- \(k: \) code for \(e_3\)

**While** \((e_1, e_2)\)

- \(m: \) code for \(e_1\)
- TEST \(k\)
- code for \(e_2\)
- GOTO \(m\)
- \(k: \)

If \(? = 0\) Then 17 else 21 end

<table>
<thead>
<tr>
<th>interp_2</th>
<th>interp_3</th>
<th>interp_3 (loaded)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH UNIT; UNARY READ; PUSH 0; OPER EQI; TEST( [PUSH 17], [PUSH 21] )</td>
<td>PUSH UNIT; UNARY READ; PUSH 0; OPER EQI; TEST L0; PUSH 17; GOTO L1; LABEL L0; PUSH 21; LABEL L1; HALT</td>
<td>0: PUSH UNIT; 1: UNARY READ; 2: PUSH 0; 3: OPER EQI; 4: TEST L0 = 7; 5: PUSH 17; 6: GOTO L1 = 9; 7: LABEL L0; 8: PUSH 21; 9: LABEL L1; 10: HALT</td>
</tr>
</tbody>
</table>

Symbolic code locations

Numeric code locations
Implement inter_2 in interp_3

```
let step = function
| ((TEST(c1, c2)) :: ds, V(BOOL true) :: evs) -> (c1 @ ds, evs)
| ((TEST(c1, c2)) :: ds, V(BOOL false) :: evs) -> (c2 @ ds, evs) ->

let step (cp, evs) = match (get_instruction cp, evs) with
| (TEST(_, Some _), V(BOOL true) :: evs) -> (cp + 1, evs)
| (TEST(_, Some i), V(BOOL false) :: evs) -> (i, evs)
| (LABEL i, evs) -> (cp + 1, evs)
| (GOTO(_, Some i), evs) -> (i, evs)
```

 interp_2.ml

Code locations are represented as

(`L`, None) : not yet loaded (assigned numeric address)

(`L`, Some i) : label “L” has been assigned numeric address i

Tricky bits again!

```
let step = function
| (POP :: ds, s :: evs) -> (ds, evs)
| (SWAP :: ds, s1 :: s2 :: evs) -> (ds, s2 :: s1 :: evs)
| ((BIND x) :: ds, (V v) :: evs) -> (ds, (V (mk_fun(c, evs_to_env evs))) :: evs)
| ((MK_CLOSURE c) :: ds, evs) -> (ds, V(mk_fun(c, evs_to_env evs))) :: evs)
| (APPLY :: ds, V(CLOSURE(_, (c, env))) :: (V v) :: evs) -> (c @ ds, (V v) :: (EV env) :: evs)

let step (cp, evs) = match (get_instruction cp, evs) with
| (POP, s :: evs) -> (cp + 1, evs)
| (SWAP, s1 :: s2 :: evs) -> (cp + 1, s2 :: s1 :: evs)
| (BIND x, (V v) :: evs) -> (cp + 1, (EV[(x, v)]) :: evs)
| (MK_CLOSURE loc, evs) -> (cp + 1, V(CLOSURE(loc, evs_to_env evs))) :: evs)
| (RETURN, (V v) :: _ :: (RA i) :: evs) -> (i, (V v) :: evs)
| (APPLY, V(CLOSURE(_, Some i), env)) :: (V v) :: evs)
```

interp_3.ml

Note that in interp_2 the body of a closure is consumed from the code stack. But in interp_3 we need to save the return address on the stack (here i is the location of the closure’s code).
Tricky bits again!

let rec compile = function
| Lambda(x, e) -> [MK_CLOSURE((BIND x) :: (compile e) @ [SWAP; POP]])
| App(e1, e2) -> (compile e2) @ (compile e1) @ [APPLY; SWAP; POP]

let rec comp = function
| App(e1, e2) ->
  let (defs1, c1) = comp e1 in
  let (defs2, c2) = comp e2 in
  (defs1 @ defs2, c2 @ c1 @ [APPLY])
| Lambda(x, e) ->
  let (defs, c) = comp e in
  let f = new_label () in
  let def = [LABEL f; BIND x] @ c @ [SWAP; POP; RETURN] in
  (def @ defs, [MK_CLOSURE((f, None))])

let compile e =
  let (defs, c) = comp e in
  c (* body of program *)
  @ [HALT] (* stop the interpreter *)
  @ defs (* function definitions *)

Interpreter 3
(very similar to interpreter 2)

let step (cp, evs) =
  match get_instruction op, evs with
  | (PUSH v, evs) -> (op + 1, (V v) :: evs)
  | (POP, s :: evs) -> (op + 1, evs)
  | (SWAP, s1 :: s2 :: evs) -> (op + 1, s2 :: s1 :: evs)
  | (BIND x, (V v) :: evs) -> (op + 1, EV[({x, v})] :: evs)
  | (LOOKUP x, evs) -> (op + 1, V(search(evs, x)) :: evs)
  | UNARY op, (V v) :: evs -> (op + 1, V(do_unary(op, v)) :: evs)
  | (GEP op, (V v1) :: evs) -> (op + 1, V(do_gep(op, v1, v2)) :: evs)
  | (NEQ v1, (V v2) :: (V v1) :: evs) -> (op + 1, [PAIR v1, v2]) :: evs)
  | (FST, V(PAIR {v, v}) :: evs) -> (op + 1, (V v) :: evs)
  | (SND, V(PAIR {v, v}) :: evs) -> (op + 1, (V v) :: evs)
  | (V v) :: evs -> (op + 1, (V v) :: evs)
  | (V v1) :: evs -> (op + 1, (V v1) :: evs)
  | (V v) :: evs -> (op + 1, (V v) :: evs)
  | (NHEQ v1, (V v1) :: evs) -> (op + 1, (V v) :: evs)
  | (CASE _, Some l, V(INL v1) :: evs) -> (op + 1, (V v1) :: evs)
  | (CASE _, Some l, V(INR v1) :: evs) -> (op + 1, (V v1) :: evs)
  | (TEST l, Some l, V(BOOL true) :: evs) -> (op + 1, evs)
  | (TEST l, Some l, V(BOOL false) :: evs) -> (op + 1, evs)
  | (ASSIGN, (V v) :: (V (REF a)) :: evs) -> (hasp.(a) <- v; {cp + 1, V(UNIT) :: evs})
  | (DEREF, (V v) :: evs) -> (op + 1, V(hasp.(a)) :: evs)
  | (NEQ v, (V v) :: evs) -> (op + 1, V(hasp.(a)) :: evs)
  | (LC_CLAUSE loc, (V v) :: (V (REF a)) :: evs) -> (cp + 1, V(hasp.(a)) :: evs)
  | (APPLY, V(CLOUSE {{l, Some l}, env}) :: (V v) :: evs) -> (cp + 1, V(hasp.(a)) :: evs)
  | (RETURN, (V v) :: evs) -> (l, (V v) :: evs)
  | (LABEL l, evs) -> (op + 1, evs)
  | (HALT, evs) -> (cp, evs)
  | (GOTO _, Some l, evs) -> (l, evs)
  _ -> compile (* "step: bad state = " ^ (string_of_state (cp, evs)) ^ "\n"*)
Some observations

• A very clean machine!
• But it still has a very inefficient treatment of environments.
• Also, pushing complex values on the stack is not what most virtual machines do. In fact, we are still using OCaml’s runtime memory management to manipulate complex values.

Example: Compiled code for rev_pair.slang

```ocaml
let rev_pair (p : int * int) : int * int = (snd p, fst p)
in
  rev_pair (21, 17)
end
```

```
MK_CLOSURE(
  BIND p; LOOKUP p; SND;
  LOOKUP p; FST; MK_PAIR;
  SWAP; POP));
BIND rev_pair;
PUSH 21;
PUSH 17;
MK_PAIR;
LOOKUP rev_pair;
APPLY;
SWAP;
POP;
SWAP;
POP
)
```

```
LABEL rev_pair
BIND p
LOOKUP p
SND
LOOKUP p
FST
MK_PAIR
SWAP
POP
POP
RETURN
```

DEMO TIME!!!
LECTURES 6
Deriving The Jargon VM
(interpreter 4)

1. First change: Introduce an **addressable stack**.
2. Replace variable lookup by a (relative) location on the stack or heap determined at **compile time**.
3. Relative to what? A **frame pointer** (fp) pointing into the stack is needed to keep track of the current **activation record**.
4. Second change: Optimise the representation of closures so that they contain **only** the values associated with the **free variables** of the closure and a pointer to code.
5. Third change: Restrict values on stack to be simple (ints, bools, heap addresses, etc). Complex data is moved to the heap, leaving pointers into the heap on the stack.
6. How might things look different in a language without first-class functions? In a language with multiple arguments to function calls?

---

**Jargon Virtual Machine**

- Stack pointer (sp)
- Frame Pointer (fp)
- Stack (really array)
- Frame 0
- Frame 1
- Frame 2
- Code (array of instructions)
- Heap (array of heap values)
- Code pointer (cp)
- Heap [0]
- Heap [heal_limit]

Need for fp to be explained soon …
The stack in interpreter 3

A stack in interpreter 3

```
(1, (2, 17))
Inl(inr(99))
```

Stack elements in interpreter 3 are not of fixed size.

Virtual machines (JVM, etc) typically restrict stack elements to be of a fixed size.

We need to shift data from the high-level stack of interpreter 3 to a lower-level stack with fixed size elements.

Solution: put the data in the heap. Place pointers to the heap on the stack.

“All problems in computer science can be solved by another level of indirection, except of course for the problem of too many indirections.”

--- David Wheeler

The Jargon VM stack

Stack

```
c
b
:
:
```

Some stack elements represent pointers into the heap

```
a : Header 2, INR
a+1 : 99
b : Header 2, INL
b+1 : a
:
:
c : Header 3, PAIR
c+1 : 1
c+2 : d
d : Header 3, PAIR
d+1 : 2
d+2 : 17
```

Heap
### interp_3.mli

```haskell
type instruction =
| PUSH of value
| LOOKUP of Ast.var
| UNARY of Ast.ordinal
| OPER of Ast.operand
| ASSIGN
| SWAP
| POP
| BIND of Ast.var
| FST
| SND
| DEREF
| APPLY
| RETURN
| MK_PAIR
| MK_INL
| MK_INR
| MK_REF
| MK_CLOSURE of location
| TEST of location
| CASE of location
| GOTO of location
| LABEL of label
| HALT
```

### jargon.mli

```haskell
type instruction =
| PUSH of stack_item (* modified *)
| LOOKUP of value_path (* modified *)
| UNARY of Ast.unary_oper
| OPER of Ast.oper
| ASSIGN
| SWAP
| POP
| (* | BIND of var not needed *)
| FST
| SND
| DEREF
| APPLY
| RETURN
| MK_PAIR
| MK_INL
| MK_INR
| MK_REF
| MK_CLOSURE of location * int (* modified *)
| TEST of location
| CASE of location
| GOTO of location
| LABEL of label
| HALT
```

### A word about implementation

#### Interpreter 3

```haskell
type value =
| REF of address
| INT of int
| BOOL of bool
| UNIT
| PAIR of value * value
| INL of value
| INR of value
| CLOSURE of location * env
```

```haskell
type env_or_value =
| EV of env
| V of value
| RA of address
```

```haskell
type env_value_stack = env_or_value list
```

#### Jargon VM

```haskell
type stack_item =
| STACK_INT of int
| STACK_BOOL of bool
| STACK_UNIT
| STACK_HI of heap_index (* Heap Index *)
| STACK_RA of code_index (* Return Address *)
| STACK_FP of stack_index (* (saved) Frame Pointer *)
```

```haskell
type heap_item =
| HEAP_INT of int
| HEAP_BOOL of bool
| HEAP_UNIT
| HEAP_HI of heap_index (* Heap Index *)
| HEAP_CI of code_index (* Code pointer for closures *)
| HEAP_HEADER of int * heap_type (* int is number items in heap block *)
```

The headers will be essential for garbage collection!
**MK_INR (MK_INL is similar)**

In interpreter 3

\[(\text{MK\_INR}, (V \ v) :: \text{evs}) \rightarrow (cp + 1, V(\text{INR}(v)) :: \text{evs})\]

**Jargon VM**

The stack before

<table>
<thead>
<tr>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
</tr>
<tr>
<td>:</td>
</tr>
</tbody>
</table>

\[\text{MK\_INR}\]

The stack after

<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
</tr>
<tr>
<td>:</td>
</tr>
</tbody>
</table>

\[a :\]

<table>
<thead>
<tr>
<th>a+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
</tr>
</tbody>
</table>

**Newly allocated locations in the heap**

<table>
<thead>
<tr>
<th>Header 2, INR</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
</tr>
</tbody>
</table>

Note: The header types are not really required. We could instead add an extra field here (for example, 0 or 1). However, header types aid in understanding the code and traces of runtime execution.

**CASE (TEST is similar)**

\[(\text{CASE } (\_ , \text{Some } \_), V(\text{INL} \ v)::\text{evs}) \rightarrow (cp + 1, (V \ v) :: \text{evs})\]
\[(\text{CASE } (\_ , \text{Some } i), V(\text{INR} \ v)::\text{evs}) \rightarrow (i, (V \ v) :: \text{evs})\]

\[
\begin{align*}
\text{cp} = t & \\
\text{CASE } i & \\
\end{align*}
\]

\[
\begin{array}{c|c}
\text{a} & \text{INR} \\
: & v \\
: & : \\
: & : \\
\end{array}
\]

\[a :\]

<table>
<thead>
<tr>
<th>a+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c}
\text{v} & \\
: & : \\
: & : \\
\end{array}
\]

\[
\begin{align*}
\text{cp} = i & \\
\end{align*}
\]

\[
\begin{array}{c|c}
\text{v} & \\
: & : \\
: & : \\
\end{array}
\]

\[
\begin{align*}
\text{cp} = t + 1 & \\
\text{CASE } i & \\
\end{align*}
\]

\[
\begin{array}{c|c}
\text{v} & \\
: & : \\
: & : \\
\end{array}
\]
MK_PAIR

In interpreter 3:

\[(\text{MK\_PAIR, } (V \ v2) :: (V \ v1) :: \text{evs}) \rightarrow (cp + 1, V(\text{PAIR}(v1, v2)) :: \text{evs})\]

In Jargon VM:

The stack before

\[
\begin{array}{c}
\text{v2} \\
\text{v1} \\
: : \\
: : \\
\end{array}
\]

The stack after

\[
\begin{array}{c}
\text{a} \\
: : \\
: : \\
\end{array}
\rightarrow
\begin{array}{c}
a : \\
: : \\
: : \\
\end{array}
\]

Newly allocated locations in the heap

\[
\begin{array}{c}
\text{Header 3, PAIR} \\
\text{v1} \\
\text{v2} \\
\end{array}
\]

FST (similar for SND)

In interpreter 3:

\[(\text{FST, } V(\text{PAIR}(v1, v2)) :: \text{evs}) \rightarrow (cp + 1, v1 :: \text{evs})\]

In Jargon VM:

The stack before

\[
\begin{array}{c}
a \\
: : \\
: : \\
\end{array}
\]

Somewhere in the heap

\[
\begin{array}{c}
\text{Header 3, PAIR} \\
\text{v1} \\
\text{v2} \\
\end{array}
\rightarrow
\begin{array}{c}
v1 \\
: : \\
: : \\
\end{array}
\]

The stack after

\[
\begin{array}{c}
\text{v1} \\
: : \\
: : \\
\end{array}
\]

Note that v1 could be a simple value (int or bool), or another heap address.
These require more care ...

In interpreter 3:

```plaintext
let step (cp, evs) =
  match (get_instruction cp, evs) with
  | (MK_CLOSURE loc, evs) -> (cp + 1, V(CLOSURE(loc, evs_to_env evs)) :: evs)
  | (APPLY, V(CLOSURE ((_, Some i), env)) :: (V v) :: evs)
    -> (i, (V v) :: (EV env) :: (RA (cp + 1)) :: evs)
  | (RETURN, (V v) :: _ :: (RA i) :: evs) -> (i, (V v) :: evs)
```

**MK_CLOSURE(c, n)**

- **c** = code location of start of instructions for closure,
- **n** = number of free variables in the body of closure.

Put values associated with free variables on stack, then construct the closure on the heap.

---

The stack before:  

<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
<th>vn</th>
</tr>
</thead>
</table>

The stack after:

| a  | a+1 | a+2 | a+n+1 |

Newly allocated locations in the heap:

<table>
<thead>
<tr>
<th>closure header</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
</tr>
<tr>
<td>v1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>vn</td>
</tr>
</tbody>
</table>
A stack frame

Return address
Saved frame pointer
Pointer to closure
Argument value

Stack frame. (Boundary may vary in the literature.)

Currently executing code for the closure at heap address “a” after it was applied to argument v.

APPLY

Interpreter 3:

\[
\text{APPLY, } \text{V(CLOSURE ((_, Some i), env)) :: (V v) :: evs)}
\]

\[
\rightarrow (i, (V v) :: (EV env) :: (RA (cp + 1)) :: evs)
\]

Jargon VM:

BEFORE
\[
\begin{align*}
\text{cp} &= k \\
\text{fp} &= j \\
\end{align*}
\]

AFTER
\[
\begin{align*}
\text{cp} &= i \\
\text{fp} &= m \\
\end{align*}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{a} \\
\vdots \\
\text{v}
\end{array} \\
\begin{array}{c}
a+1 \\
\vdots \\
a+2 \\
\vdots \\
a+n+1
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{a} \\
\vdots \\
\text{v}
\end{array} \\
\begin{array}{c}
\text{I} \\
\vdots \\
\text{v1}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]

\[
\begin{array}{c}
k+1 \\
\vdots \\
\text{fp}
\end{array}
\]
Interpreter 3:

\( (\text{RETURN}, (V \ v) :: _ :: (\text{RA } i) :: \text{evs}) \rightarrow (i, (V \ v) :: \text{evs}) \)

**Jargon VM:**

**BEFORE**

\[
\begin{array}{c}
\text{cp} = i \\
\text{v2} \\
\text{t} \\
\text{j} \\
\text{a} \\
\text{v1} \\
\vdots \\
\vdots
\end{array}
\]

**AFTER**

\[
\begin{array}{c}
\text{cp} = t \\
\text{v2} \\
\vdots \\
\vdots
\end{array}
\]

Replace stack frame with return value

( return address)

**Finding a variable’s value at runtime**

Suppose we are executing code associated with this closure. Then every free variable in the body of the closure can be found from the frame pointer \( \text{fp} : \)

- Formal parameter: at stack location \( \text{fp} - 2 \)
- Other free variables:
  - Follow heap pointer found at \( \text{fp} - 1 \)
  - Each free variable can be associated with a **fixed offset** from this heap address
LOOKUP (HEAP_OFFSET k)

Interpreter 3:

(LOOKUP x, evs) -> (cp + 1, V(search(evs, x)) :: evs)

Jargon VM:

BEFORE

sp  FREE
k+1   j
    a
    v

fp  a:
    Header
    i
    v1
    : : vk
    : :

AFTER

sp  FREE
vk
    : :
k+1   j
    a
    v

fp  : :
    : :
    : :
    : :

push argument value onto the stack

LOOKUP (STACK_OFFSET -2)

Interpreter 3:

(LOOKUP x, evs) -> (cp + 1, V(search(evs, x)) :: evs)

Jargon VM:

BEFORE

sp  FREE
k+1   j
    a
    v

fp  a:
    Header
    i
    v1
    : : vk
    : :

AFTER

sp  FREE
v
    : :
k+1   j
    a
    v

fp  : :
    : :
    : :
    : :
Oh, one problem

```
let rec comp = function
    : | LetFun(f, (x, e1), e2) ->
       let (defs1, c1) = comp e1 in
       let (defs2, c2) = comp e2 in
       let def = [LABEL f; BIND x] @ c1 @ [SWAP; POP; RETURN] in
       (def @ defs1 @ defs2,
        [MK_CLOSURE((f, None)); BIND f] @ c2 @ [SWAP; POP])
```

Problem: Code c2 can be anything --- how are we going to find the closure for f when we need it? It has to be a fixed offset from a frame pointer --- we no longer scan the stack for bindings!

```
let rec comp vmap = function
    : | LetFun(f, (x, e1), e2) -> comp vmap (App(Lambda(f, e2), Lambda(x, e1)))
```

Similar trick for LetRecFun

---

**LOOKUP (STACK_OFFSET -1)**

For recursive function calls, push current closure on to the stack.

**Jargon VM:**

**BEFORE**

```
sp
FREE
: : 
: : 
: : 
k+1
j
a
v
: : 
: : 
```

**AFTER**

```
sp
FREE
a
: : 
: : 
k+1
j
a
v
: : 
: : 
```

**LOOKUP (STACK_OFFSET -1)**

```
closure
```

```
closure
```

**fp**

```
```
Example: Compiled code for rev_pair.slang

```haskell
let rev_pair (p : int * int) : int * int = (snd p, fst p) in
  rev_pair (21, 17)
end
```

After the front-end, compile treats this as follows.

```haskell
App(
  Lambda(
    "rev_pair",
    App(Var "rev_pair", Pair (Integer 21, Integer 17))),
  Lambda("p", Pair(Snd (Var "p"), Fst (Var "p"))))
```

Example: Compiled code for rev_pair.slang

```latex
\begin{verbatim}
MK_CLOSURE(L1, 0)
MK_CLOSURE(L0, 0)
APPLY
HALT
L0 : PUSH STACK_INT 21
PUSH STACK_INT 17
MK_PAIR
LOOKUP STACK_LOCATION -2
APPLY
RETURN
L1 : LOOKUP STACK_LOCATION -2
SND
LOOKUP STACK_LOCATION -2
FST
MK_PAIR
RETURN
\end{verbatim}
```

-- Make closure for second lambda
-- Make closure for first lambda
-- do application
-- the end!
-- code for first lambda, push 21
-- push 17
-- make the pair on the heap
-- push closure for second lambda on stack
-- apply first lambda
-- return from first lambda
-- code for second lambda, push arg on stack
-- extract second part of pair
-- push arg on stack again
-- extract first part of pair
-- construct a new pair
-- return from second lambda
Example : trace of rev_pair.slang execution

Installed Code =

0: MK_CLOSURE(L1 = 11, 0)
1: MK_CLOSURE(L0 = 4, 0)
2: APPLY
3: HALT
4: LABEL L0
5: PUSH STACK_INT 21
6: PUSH STACK_INT 17
7: MK_PAIR
8: LOOKUP STACK_LOCATION-2
9: APPLY
10: RETURN
11: LABEL L1
12: LOOKUP STACK_LOCATION-2
13: SND
14: LOOKUP STACK_LOCATION-2
15: FST
16: MK_PAIR
17: RETURN

========== state 1 ==========
cp = 0 -> MK_CLOSURE(L1 = 11, 0)
f0 = 0
Stack =
1: STACK_RA 0
0: STACK_FP 0

========== state 2 ==========
cp = 1 -> MK_CLOSURE(L0 = 4, 0)
f0 = 0
Stack =
2: STACK_HI 0
1: STACK_RA 0
0: STACK_FP 0

========== state 15 ==========
cp = 16 -> MK_PAIR
fp = 8
Stack =
11: STACK_INT 21
10: STACK_INT 17
9: STACK_RA 10
8: STACK_FP 4
7: STACK_HI 0
6: STACK_HI 4
5: STACK_RA 3
4: STACK_FP 0
3: STACK_HI 2
2: STACK_HI 0
1: STACK_RA 0
0: STACK_FP 0

Heap =
0 -> HEAP_HEADER(2, HT_CLOSURE)
1 -> HEAP_CI 11
2 -> HEAP_HEADER(2, HT_CLOSURE)
3 -> HEAP_CI 4
4 -> HEAP_HEADER(3, HT_PAIR)
5 -> HEAP_INT 21
6 -> HEAP_INT 17

========== state 19 ==========
cp = 3 -> HALT
fp = 0
Stack =
2: STACK_HI 7
1: STACK_RA 0
0: STACK_FP 0

Heap =
0 -> HEAP_HEADER(2, HT_CLOSURE)
1 -> HEAP_CI 11
2 -> HEAP_HEADER(2, HT_CLOSURE)
3 -> HEAP_CI 4
4 -> HEAP_HEADER(3, HT_PAIR)
5 -> HEAP_INT 21
6 -> HEAP_INT 17
7 -> HEAP_HEADER(3, HT_PAIR)
8 -> HEAP_INT 17
9 -> HEAP_INT 21

Jargon VM :
output> (17, 21)
Example: closure_add.slang

```plaintext
let f(y : int) : int -> int = let g(x : int) : int = y + x in g end
in let add21 : int -> int = f(21)
in let add17 : int -> int = f(17)
in add17(3) + add21(10)
end
end
define f(y : int) = let g(x : int) = y + x in g
end
end

After the front-end, this becomes represented as follows.

```

MK_CLOSURE(L3, 0)
MK_CLOSURE(L0, 0)
APPLY
HALT
L0 : PUSH STACK_INT 21
LOOKUP STACK_LOCATION -2
APPLY
LOOKUP STACK_LOCATION -2
MK_CLOSURE(L1, 1)
APPLY
RETURN
L1 : PUSH STACK_INT 17
LOOKUP HEAP_LOCATION 1
APPLY
LOOKUP STACK_LOCATION -2
MK_CLOSURE(L2, 1)
APPLY
RETURN
L2 : PUSH STACK_INT 3
LOOKUP STACK_LOCATION -2
APPLY
L3 : PUSH STACK_INT 10
LOOKUP HEAP_LOCATION 1
APPLY
OPER ADD
RETURN
L4 : LOOKUP STACK_LOCATION -2
MK_CLOSURE(L5, 1)
MK_CLOSURE(L4, 0)
APPLY
RETURN
L5 : LOOKUP HEAP_LOCATION 1
LOOKUP STACK_LOCATION -2
OPER ADD
RETURN
```

Can we make sense of this?

```plaintext
MK_CLOSURE(L3, 0)
MK_CLOSURE(L0, 0)
APPLY
HALT
L0 : PUSH STACK_INT 21
LOOKUP STACK_LOCATION -2
APPLY
LOOKUP STACK_LOCATION -2
MK_CLOSURE(L1, 1)
APPLY
RETURN
L1 : PUSH STACK_INT 17
LOOKUP HEAP_LOCATION 1
APPLY
LOOKUP STACK_LOCATION -2
MK_CLOSURE(L2, 1)
APPLY
RETURN
L2 : PUSH STACK_INT 3
LOOKUP STACK_LOCATION -2
APPLY
L3 : PUSH STACK_INT 10
LOOKUP HEAP_LOCATION 1
APPLY
OPER ADD
RETURN
L4 : LOOKUP STACK_LOCATION -2
MK_CLOSURE(L5, 1)
MK_CLOSURE(L4, 0)
APPLY
RETURN
L5 : LOOKUP HEAP_LOCATION 1
LOOKUP STACK_LOCATION -2
OPER ADD
RETURN
```

Note: we really do need closures on the heap here—the values 21 and 17 do not exist on the stack at this point in the execution.
The Gap, illustrated

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

Taking stock

Starting from a direct implementation of Slang/L3 semantics, we have DERIVED a Virtual Machine in a step-by-step manner. The correctness of each step is (more or less) easy to check.
Remarks

1. The semantic GAP between a Slang/L3 program and a low-level translation (say x86/Unix) has been significantly reduced.


3. However, using a lower-level implementation (say x86, exploiting fast registers) to generate very efficient code is not so easy. See Part II Optimising Compilers.

Verification of compilers is an active area of research. See CompCert, CakeML, and DeepSpec.

We could implement a Jargon byte code interpreter ...

```c
void vsm_execute_instruction(vsm_state *state, bytecode instruction) {
    opcode code   = instruction.code;
    argument arg1 = instruction.arg1;
    switch (code) {
        case PUSH: { state->stack[state->sp++] = arg1; state->pc++; break; }
        case POP : { state->sp--; state->pc++; break; }
        case GOTO: { state->pc = arg1; break; }
        case STACK_LOOKUP: {
            state->stack[state->sp++] =
                state->stack[state->fp + arg1];
            state->pc++;  break; }
        ...
    }
}
```

- Generate compact byte code for each Jargon instruction.
- Compiler writes byte codes to a file.
- Implement an interpreter in C or C++ for these byte codes.
- Execution is much faster than our jargon.ml implementation.
- Or, we could generate assembly code from Jargon instructions ....
One of the great benefits of Virtual Machines is their portability. However, for more efficient code we may want to compile to assembler. Lost portability can be regained through the extra effort of implementing code generation for every desired target platform.

Lectures 7 --- 12
Assorted Topics

1. Separate compilation, linking
2. Interface with OS
3. Stacks vs registers
4. Calling conventions
5. Generating assembler code
6. Simple optimisations
7. The runtime system (automatic memory management, …)
8. Static links (for languages without nested functions/procedures)
9. Implementing OOP with inheritance
10. Implementing exceptions
11. Compiling a compiler, “boot strapping”
Assembly and Linking

From symbolic names and addresses to numeric codes and numeric addresses

Name resolution, create single address space by address relocation

The gcc manual (810 pages)
https://gcc.gnu.org/onlinedocs/gcc-5.3.0/gcc.pdf

Chapter 9: Binary Compatibility

9 Binary Compatibility

Binary compatibility encompasses several related concepts:
application binary interface (ABI)

The set of runtime conventions followed by all of the tools that deal with binary representations of a program, including compilers, assemblers, linkers, and language runtime support. Some ABIs are formal with a written specification, possibly designed by multiple interested parties. Others are simply the way things are actually done by a particular set of tools.
Applications Binary Interface (ABI)

We will use x86/Unix as our running example. Specifies many things, including the following.

- C calling conventions used for systems calls or calls to compiled C code.
  - Register usage and stack frame layout
  - How parameters are passed, results returned
  - Caller/callee responsibilities for placement and cleanup
- Byte-level layout and semantics of object files.
  - Executable and Linkable Format (ELF). Formerly known as Extensible Linking Format.
- Linking, loading, and name mangling

Note: the conventions are required for portable interaction with compiled C. Your compiled language does not have to follow the same conventions!

Object files

Must contain at least

- Program instructions
- Symbols being exported
- Symbols being imported
- Constants used in the program (such as strings)

Executable and Linkable Format (ELF) is a common format for both linker input and output.
ELF details (1)

<table>
<thead>
<tr>
<th>Header information; positions and sizes of sections</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.text</code> segment (code segment): binary data</td>
</tr>
<tr>
<td><code>.data</code> segment: binary data</td>
</tr>
<tr>
<td><code>.rela.text</code> code segment relocation table: list of (offset, symbol) pairs giving:</td>
</tr>
<tr>
<td>(i) offset within <code>.text</code> to be relocated; and</td>
</tr>
<tr>
<td>(iii) by which symbol</td>
</tr>
<tr>
<td><code>.rela.data</code> data segment relocation table: list of (offset, symbol) pairs giving:</td>
</tr>
<tr>
<td>(i) offset within <code>.data</code> to be relocated; and</td>
</tr>
<tr>
<td>(iii) by which symbol</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

ELF details (2)

| ... |
| `.symtab` symbol table: |
| List of external symbols (as triples) used by the module. |
| Each is (attribute, offset, symname) with attribute: |
| 1. `undef`: externally defined, offset is ignored; |
| 2. defined in code segment (with offset of definition); |
| 3. defined in data segment (with offset of definition). |
| Symbol names are given as offsets within `.strtab` to keep table entries of the same size. |
| `.strtab` string table: |
| the string form of all external names used in the module |
The (Static) Linker

What does a linker do?
• takes some object files as input, notes all undefined symbols.
• recursively searches libraries adding ELF files which define such symbols until all names defined (“library search”).
• whinges if any symbol is undefined or multiply defined.

Then what?
• concatenates all code segments (forming the output code segment).
• concatenates all data segments.
• performs relocations (updates code/data segments at specified offsets).

Dynamic vs. Static linking

Static linking (compile time)
Problem: a simple “hello world” program may give a 10MB executable if it refers to a big graphics or other library.

Dynamic linking (run time)
For shared libraries, the object files contain stubs, not code, and the operating system loads and links the code on demand.

Pros and Cons of dynamic linking:

(+) Executables are smaller
(+ ) Bug fixes to libraries don't require re-linking.
(-) Non-compatible changes to a library can wreck previously working programs (“dependency hell”).
A “runtime system”

A library implementing functionality needed to run compiled code on a given operating system. Normally tailored to the language being compiled.

- Implements interface between OS and language.
- May implement memory management.
- May implement “foreign function” interface (say we want to call compiled C code from Slang code, or vice versa).
- May include efficient implementations of primitive operations defined in the compiled language.
- For some languages, the runtime system may perform runtime type checking, method lookup, security checks, and so on.
- …

Runtime system

Targeting a VM

- Generated code
  - Virtual Machine
    - Implementation
      - Includes runtime system

Targeting a platform

- Generated code
  - Run-time system
  - Linker
    - Executable

In either case, implementers of the compiler and the runtime system must agree on many low-level details of memory layout and data representation.
**Typical (Low-Level) Memory Layout (UNIX)**

Rough schematic of traditional layout in (virtual) memory.

- **Stack**: The heap is used for dynamically allocating memory. Typically either for very large objects or for those objects that are returned by functions/procedures and must outlive the associated activation record.
- **Heap**: In languages like Java and ML, the heap is managed automatically ("garbage collection").

**Stack vs registers**

- **Stack-oriented**: (+) argument locations is implicit, so instructions are smaller. (-) Execution is slower
- **Register-oriented**: (+++) Execution MUCH faster (-) argument location is explicit, so instructions are larger
Main dilemma: registers are fast, but are fixed in number. And that number is rather small.

- Manipulating the stack involves RAM access, which can be orders of magnitude slower than register access (the “von Neumann Bottleneck”)
- Fast registers are (today) a scarce resource, shared by many code fragments
- How can registers be used most effectively?
  - Requires a careful examination of a program's structure
  - Analysis phase: building data structures (typically directed graphs) that capture definition/use relationships
  - Transformation phase: using this information to rewrite code, attempting to most efficiently utilise registers
  - Problem is NP-complete
  - One of the central topics of Part II Optimising Compilers.
- Here we focus **only** on general issues: calling conventions and register spilling

---

**Caller/callee conventions**

- Caller and callee code may use overlapping sets of registers
- An agreement is needed concerning use of registers
  - Are some arguments passed in specific registers?
  - Is the result returned in a specific register?
  - If the caller and callee are both using a set of registers for “scratch space” then caller or callee must save and restore these registers so that the caller’s registers are not obliterated by the callee.
- Standard calling conventions identify specific subsets of registers as “caller saved” or “callee saved”
  - **Caller saved**: if caller cares about the value in a register, then must save it before making any call
  - **Callee saved**: The caller can be assured that the callee will leave the register intact (perhaps by saving and restoring it)
Another C example.  
X86, 64 bit, with gcc

```c
int callee(int, int, int, int, int, int, int, int);

int caller(void)
{
    int ret;
    ret = callee(1, 2, 3, 4, 5, 6, 7);
    ret += 5;
    return ret;
}
```

_regsiter spilling

- What happens when all registers are in use?
- Could use the stack for scratch space …
- … or (1) move some register values to the stack, (2) use the registers for computation, (3) restore the registers to their original value
- This is called register spilling
## A Crash Course in x86 assembler

- A CISC architecture
- There are 16, 32 and 64 bit versions
- 32 bit version:
  - General purpose registers: EAX EBX ECX EDX
  - Special purpose registers: ESI EDI EBP EIP ESP
    - EBP: normally used as the frame pointer
    - ESP: normally used as the stack pointer
    - EDI: often used to pass (first) argument
    - EIP: the code pointer
  - Segment and flag registers that we will ignore …
- 64 bit version:
  - Rename 32-bit registers with “R” (RAX, RBX, RCX, …)
  - More general registers: R8 R9 R10 R11 R12 R13 R14 R15

<table>
<thead>
<tr>
<th>Register names can indicate “width” of a value.</th>
<th>rax: 64 bit version</th>
<th>eax: 32 bit version (or lower 32 bits of rax)</th>
<th>ax: 16 bit version (or lower 16 bits of eax)</th>
<th>al: lower 8 bits of ax</th>
<th>ah: upper 8 bits of ax</th>
</tr>
</thead>
</table>

### See https://en.wikibooks.org/wiki/X86_Assembly

The syntax of x86 assembler comes in several flavours. Here are two examples of “put integer 4 into register eax”:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $4, %eax</td>
<td>GAS (aka AT&amp;T) notation</td>
</tr>
<tr>
<td>mov eax, 4</td>
<td>Intel notation</td>
</tr>
</tbody>
</table>

I will (mostly) use the GAS syntax, where a suffix is used to indicate width of arguments:
- b (byte) = 8 bits
- w (word) = 16 bits
- l (long) = 32 bits
- q (quad) = 64 bits

For example, we have movb, movw movl, and movq.
Examples (in GAS notation)

- `movl $4, %eax` # put 32 bit integer 4 in register eax
- `movw $4, %eax` # put 16 bit integer 4 in lower 16 bits of eax
- `movb $4, %eax` # put 4 bit integer 4 in lowest 4 bits of eax
- `movl %esp, %ebp` # put the contents of esp into ebp
- `movl (%esp), %ebp` # interpret contents of esp as a memory address. Copy the value at that address into register ebp
- `movl %esp, (%ebp)` # interpret contents of ebp as a memory address. Copy the value in esp to that address.
- `movl %esp, 4(%ebp)` # interpret contents of ebp as a memory address. Add 4 to that address. Copy the value in esp to this new address.

A few more examples

- `call label` # push return address on stack and jump to label
- `ret` # pop return address off stack and jump there
- `subl $4, %esp` # subtract 4 from esp. That is, adjust the stack pointer to make room for one 32-bit (4 byte) value. (stack grows downward!)

Assume that we have implemented a procedure in C called `allocate` that will manage heap memory. We will compile and link this in with code generated by the slang compiler. At the x86 level, `allocate` will expect a header in edi and return a heap pointer in eax.
Some Jargon VM instructions are “easy” to translate

Remember: X86 is CISC, so RISC architectures may require more instructions …

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOTO loc</td>
<td>jmp loc</td>
</tr>
<tr>
<td>POP v</td>
<td>addl $4, %esp</td>
</tr>
<tr>
<td></td>
<td>// move stack pointer 1 word = 4 bytes</td>
</tr>
<tr>
<td></td>
<td>subl $4, %esp</td>
</tr>
<tr>
<td></td>
<td>// make room on top of stack</td>
</tr>
<tr>
<td></td>
<td>movl $i, (%esp)</td>
</tr>
<tr>
<td></td>
<td>// where i is an integer representing v</td>
</tr>
<tr>
<td>PUSH v</td>
<td>movl (%esp), %edx</td>
</tr>
<tr>
<td></td>
<td>// store “a” into edx</td>
</tr>
<tr>
<td></td>
<td>movl 4(%edx), %edx</td>
</tr>
<tr>
<td></td>
<td>// load v1, 4 bytes, 1 word, after header</td>
</tr>
<tr>
<td></td>
<td>movl %edx, (%esp)</td>
</tr>
<tr>
<td></td>
<td>// replace “a” with “v1” at top of stack</td>
</tr>
<tr>
<td>FST v</td>
<td>movl (%esp), %edx</td>
</tr>
<tr>
<td></td>
<td>// store “a” into edx</td>
</tr>
<tr>
<td></td>
<td>movl 8(%edx), %edx</td>
</tr>
<tr>
<td></td>
<td>// vload v2, 8 bytes, 2 words, after header</td>
</tr>
<tr>
<td></td>
<td>movl %edx, (%esp)</td>
</tr>
<tr>
<td></td>
<td>// replace “a” with “v2” at top of stack</td>
</tr>
</tbody>
</table>

… while others require more work

One possible x86 (32 bit) implementation of MK_PAIR:

```
movl $3, %edi    // construct header in edi
shr $16, %edi,   // … put size in upper 16 bits (shift right)
movw %PAIR, %di  // … put type in lower 16 bits of edi
call allocate    // input: header in edi, output: “a” in eax
movl (%esp), %edx // move “v2” to the heap,
movl %edx, 8(%eax) // … using temporary register edx
addl $4, %esp    // adjust stack pointer (pop “v2”)
movl (%esp), %edx // move “v1” to the heap
movl %edx, 4(%eax) // … using temporary register edx
movl %eax, (%esp) // copy value “a” to top of stack
```
call function computed at runtime?

For things you don’t understand, just experiment! OK, you need to pull an address out of a closure and call it. Hmm, how does something similar get compiled from C?

```c
int func ( int (*f)(int) ) { return (*f)(17); } /* pass a function pointer and apply it */
```

```assembly
(func:
  pushq  %rbp                 # save frame pointer
  movq  %rsp, %rbp       # set frame pointer to stack pointer
  subq $16, %rsp         # make some room on stack
  movl $17, %eax        # put 17 in argument register eax
  movq %rdi, -8(%rbp)  # rdi contains the argument f
  movl %eax, %edi      # put 17 in register edi, so f will get it
  callq -8(%rbp)       # WOW, a computed address for call!
  addq $16, %rsp       # restore stack pointer
  popq %rbp)           # restore old frame pointer
  ret)
```

What about arithmetic?

Houston, we have a problem….

- It may not be obvious now, but if we want to have automated memory management we need to be able to distinguish between values (say integers) and pointers at runtime.
- Have you ever noticed that integers in SML or Ocaml are either 31 (or 63) bits rather than the native 32 (or 64) bits?
  - That is because these compilers use a the least significant bit to distinguish integers (bit = 1) from pointers (bit = 0).
  - OK, this works. But it may complicate every arithmetic operation!
  - This is another exercise left for you to ponder….
New topic: Memory Management

- Many programming languages allow programmers to (implicitly) allocate new storage dynamically, with no need to worry about reclaiming space no longer used.
  - New records, arrays, tuples, objects, closures, etc.
  - Java, SML, OCaml, Python, JavaScript, Python, Ruby, Go, Swift, SmallTalk, …
- Memory could easily be exhausted without some method of reclaiming and recycling the storage that will no longer be used.
  - Often called “garbage collection”
  - Is really “automated memory management” since it deals with allocation, de-allocation, compaction, and memory-related interactions with the OS.

Explicit (manual) memory management

- User library manages memory; programmer decides when and where to allocate and de-allocate
  - void* malloc(long n)
  - void free(void *addr)
  - Library calls OS for more pages when necessary
  - Advantage: Gives programmer a lot of control.
  - Disadvantage: people too clever and make mistakes. Getting it right can be costly. And don’t we want to automate-away tedium?
  - Advantage: With these procedures we can implement memory management for “higher level” languages ;)}
Automation is based on an approximation: if data can be reached from a root set, then it is not “garbage.”

Type information required (pointer or not), some kind of “tagging” needed.

... Identify Cells Reachable From Root Set...
... reclaim unreachable cells

But How? Two basic techniques, and many variations

- **Reference counting**: Keep a reference count with each object that represents the number of pointers to it. Is garbage when count is 0.
- **Tracing**: find all objects reachable from root set. Basically transitive close of pointer graph.

For a very interesting (non-examinable) treatment of this subject see

**A Unified Theory of Garbage Collection.**
David F. Bacon, Perry Cheng, V.T. Rajan.
OOPSLA 2004.

In that paper reference counting and tracing are presented as “dual” approaches, and other techniques are hybrids of the two.
Reference Counting, basic idea:

- Keep track of the number of pointers to each object (the reference count).
- When Object is created, set count to 1.
- Every time a new pointer to the object is created, increment the count.
- Every time an existing pointer to an object is destroyed, decrement the count.
- When the reference count goes to 0, the object is unreachable garbage.

Reference counting can’t detect cycles!

Cons
- Space/time overhead to maintain count.
- Memory leakage when have cycles in data.

Pros
- Incremental (no long pauses to collect...)
Mark and Sweep

• A two-phase algorithm
  – Mark phase: Depth first traversal of object graph from the roots to mark live data
  – Sweep phase: iterate over entire heap, adding the unmarked data back onto the free list

Copying Collection

• Basic idea: use 2 heaps
  – One used by program
  – The other unused until GC time
• GC:
  – Start at the roots & traverse the reachable data
  – Copy reachable data from the active heap (from-space) to the other heap (to-space)
  – Dead objects are left behind in from space
  – Heaps switch roles
Copying Collection

![Diagram of copying collection process]

- Pros
  - Simple & collects cycles
  - Run-time proportional to # live objects
  - Automatic compaction eliminates fragmentation

- Cons
  - Twice as much memory used as program requires
    - Usually, we anticipate live data will only be a small fragment of store
    - Allocate until 70% full
    - From-space = 70% heap; to-space = 30%
  - Long GC pauses = bad for interactive, real-time apps
**OBSERVATION: for a copying garbage collector**

- 80% to 98% new objects die very quickly.
- An object that has survived several collections has a bigger chance to become a long-lived one.
- It’s a inefficient that long-lived objects be copied over and over.

![Diagram from Andrew Appel's Modern Compiler Implementation](image)

---

**IDEA: Generational garbage collection**

Segregate objects into multiple areas by age, and collect areas containing older objects less often than the younger ones.

![Diagram from Andrew Appel's Modern Compiler Implementation](image)
Other issues...

- When do we promote objects from young generation to old generation?
  - Usually after an object survives a collection, it will be promoted.
- Need to keep track of older objects pointing to newer ones!
- How big should the generations be?
  - When do we collect the old generation?
  - After several minor collections, we do a major collection.
- Sometimes different GC algorithms are used for the new and older generations.
  - Why? Because they have different characteristics.
  - Copying collection for the new:
    - Less than 10% of the new data is usually live.
    - Copying collection cost is proportional to the live data.
  - Mark-sweep for the old.

New topic: Simple optimisations.

Inline expansion

```
fun f(x) = x + 1
fun g(x) = x - 1
...
fun h(x) = f(x) + g(x)
```

**inline f and g**

```
fun f(x) = x + 1
fun g(x) = x - 1
...
fun h(x) = (x+1) + (x-1)
```

(+) Avoid building activation records at runtime.
(+) May allow further optimisations.

(-) May lead to “code bloat” (apply only to functions with “small” bodies?)

Question: if we inline all occurrences of a function, can we delete its definition from the code? What if it is needed at link time?
Be careful with variable scope

Inline g in h

```plaintext
let val x = 1
  fun g(y) = x + y
  fun h(x) = g(x) + 1
in
  h(17)
end
```

NO

```plaintext
let val x = 1
  fun g(y) = x + y
  fun h(x) = g(x) + 1
in
  h(17)
end
```

YES

```plaintext
let val x = 1
  fun g(y) = x + y
  fun h(x) = x + y + 1
in
  h(17)
end
```

What kind of care might be needed will depend on the representation level of the Intermediate code involved.

(b) Constant propagation, constant folding

```
let x = 2
let y = x - 1
let z = y * 17
```

Propagate constants and evaluate simple expressions at compile-time

Note: opportunities are often exposed by inline expansion!

```
let x = 2
let y = 1
let z = y * 17
```

```
let x = 2
let y = 1
let z = 1 * 17
```

```
let x = 2
let y = 1
let z = 17
```

But be careful

How about this?

Replace

```
deserialize (x * 0)
```

with

```
0
```

OOPS, not if x has type float!

```
NAN*0 = NAN
```

David Gries:

“Never put off till run-time what you can do at compile-time.”
Peephole Optimization

Communications of the ACM,
July 1965

Example 1. Source code:

\[
X := Y; \\
Z := X + Z
\]

Compiled code:

- LDA Y: load the accumulator from Y
- STA X: store the accumulator in X
- LDA X: load the accumulator from X
- ADD Z: add the contents of Z
- STA Z: store the accumulator in Z

Eliminate!

Results for syntax-directed code generation.

---

Sweep a window over the code sequence looking for instances of simple code patterns that can be rewritten to better code … (might be combined with constant folding, etc, and employ multiple passes)

Examples

- eliminate useless combinations (push 0; pop)
- introduce machine-specific instructions
- improve control flow. For example: rewrite

  “GOTO L1 … L1: GOTO L2”

  to

  “GOTO L2 … L1: GOTO L2”

(c) peephole optimisation
gcc example.
-\texttt{-O}m\textgreater\ turns on optimisation to level m

\begin{verbatim}
g.c
int h(int n) { return (0 < n) ? n : 101 ; }
int g(int n) { return 12 * h(n + 17); }
g.s (fragment)
gcc -O2 -S -c g.c
  .g:
  .cfi_startproc
  pushq %rbp
  movq %rsp, %rbp
  addl $17, %edi
  imull $12, %edi, %ecx
  testl %edi, %edi
  movl $1212, %eax
  cmovgl %ecx, %eax
  popq %rbp
  ret
  .cfi_endproc
\end{verbatim}

\textbf{Wait. What happened to the call to h???

GNU AS (GAS) Syntax
x86, 64 bit

\begin{verbatim}
gcc example (-O<m> turns on optimisation)

\begin{verbatim}
g.c
int h(int n) { return (0 < n) ? n : 101 ; }
int g(int n) { return 12 * h(n + 17); }
g.s (fragment)
gcc -O2 -S -c g.c
\end{verbatim}

\textbf{The compiler must have done something similar to this:}

\begin{verbatim}
int g(int n) { return 12 * h(n + 17); }
  \rightarrow
int g(int n) { int t := n+ 17; return 12 * h(t); }
  \rightarrow
int g(int n) { int t := n+ 17; return 12 *((0 < t) ? t : 101 ); }
  \rightarrow
int g(int n) { int t := n+ 17; return (0 < t) ? 12 * t : 1212 ; }
  \rightarrow ...
\end{verbatim}

\end{verbatim}
New topic : static links on the call stack.

- Many textbooks on compilers treat only languages with first-order functions --- that is, functions cannot be passed as an argument or returned as a result. In this case, we can avoid allocating environments on the heap since all values associated with free variables will be somewhere on the stack!
- But how do we find these values? We optimise stack search by following a chain of **static links**. Static links are added to every stack frame and point to the stack frame of the last invocation of the defining function.
- One other thing: most languages take multiple arguments for a function/procedure call.

---

**Terminology: Caller and Callee**

fun f (x, y) = e1
...
fun g(w, v) = w + f(v, v)

For this invocation of the function f, we say that g is the **caller** while f is the **callee**

Recursive functions can play both roles at the same time …
Nesting depth

Pseudo-code

fun b(z) = e

fun g(x1) =
    fun h(x2) =
        fun f(x3) = e3(x1, x2, x3, b, g, h, f)
        in
            e2(x1, x2, b, g, h, f)
        end
    in
        e1(x1, b, g, h)
    end
...  b(g(17))  ...

Function g is the **definer** of h. Functions g and b must share a definer defined at depth k-1
Stack with static links and variable number of arguments

- Stack frame for **callee** defined at nesting depth \(i \leq k + 1\).
- Stack frame for **caller** defined at nesting depth \(k\) used to evaluate code at depth \(k + 1\).

The static link points down to the closest frame of **definer** at nesting depth \(i - 1\).

**caller and callee at same nesting depth** \(k\)

- Code
  - \(j: \text{call } f\)
  - \(f: \ldots\)
- Call \(f\)
- \(sp\) points to \(FREE\)
- \(fp\) points to caller's frame
- \(SL(k-1)\)

- Code
  - \(j: \text{call } f\)
  - \(f: \ldots\)
- \(sp\) points to \(FREE\)
- \(fp\) points to \(SL(k-1)\)
- \(j+1\)
**caller at depth k and callee at depth i < k**

```
cp →
j : call f
     | f : .......
     Code

sp →
     FREE
     | SL{k - 1}

fp →
     FREE
     | SL{k - 1}
```

- `p := !(fp + 2);`
- `for c = 1 to k - i {
  p := !(p + 2);
}
- `SL{i-1} := p;`

**call f (k - i)**

---

**caller at depth k and callee at depth k + 1**

```
cp →
j : call f
     | f : .......
     Code

sp →
     FREE
     | FP-saved
     j+1

fp →
     FREE
     | FP-saved
     j+1
```

- `p := !(fp + 2);`
- `for c = 1 to k - i {
  p := !(p + 2);
}
- `SL{i-1} := p;`
Access to argument values at static distance 0

Access to argument values at static distance d, 0 < d

\[ p := !(fp + 2); \]
\[ \text{for } c = 1 \text{ to } d \]
\[ \{ \]
\[ \quad p := !(p + 2); \]
\[ \} \]
\[ v := !(p - j); \]
New Topic:  
OOP Objects (single inheritance)

let start := 10

class Vehicle extends object {
    var position := start
    method move(int x) = {position := position + x}
}
class Car extends Vehicle {
    var passengers := 0
    method await(v : Vehicle) =
        if (v.position < position)
        then v.move(position - v.position)
        else self.move(10)
}
class Truck extends Vehicle {
    method move(int x) =
        if x <= 55 then position := position + x
}
var t := new Truck
var c := new Car
var v : Vehicle := c
in
    c.passengers := 2;
    c.move(60);
    v.move(70);
    c.await(t)
end

Object Implementation?

- how do we access object fields?
  - both inherited fields and fields for the current object?

- how do we access method code?
  - if the current class does not define a particular method, where do we go to get the inherited method code?
  - how do we handle method override?

- How do we implement subtyping ("object polymorphism")?
  - If B is derived from A, then need to be able to treat a pointer to a B-object as if it were an A-object.
Another OO Feature

• Protection mechanisms
  – to encapsulate local state within an object, Java has “private” “protected” and “public” qualifiers
    • private methods/fields can’t be called/used outside of the class in which they are defined
  – This is really a scope/visibility issue! Front-end during semantic analysis (type checking and so on), the compiler maintains this information in the symbol table for each class and enforces visibility rules.

Object representation

```
class A {
  public:
    int a1, a2;
    virtual void m1(int i) {
      a1 = i;
    }
    virtual void m2(int i) {
      a2 = a1 + i;
    }
}
```

An A object

object data

vtable for class A

NB: a compiler typically generates methods with an extra argument representing the object (self) and used to access object data.
Inheritance ("pointer polymorphism")

class B : public A {
    public:
        int b1;
        virtual void m3(void) {
            b1 = a1 + a2;
        }
    }

a B object

object data

vtable for class B

Note that a pointer to a B object can be treated as if it were a pointer to an A object!

Method overriding

class C : public A {
    public:
        int c1;
        virtual void m3(void) {
            b1 = a1 + a2;
        }
        virtual void m2(int i) {
            a2 = c1 + i;
        }
    }

a C object

object data

vtable for class C

declared defined
Static vs. Dynamic

- which method to invoke on overloaded polymorphic types?

```cpp
class C *c = ...;
class A *a = c;
a->m2(3);
```

Static dispatch:
- `m2_A_A(a, 3);`
- `m2_A_C(a, 3);`

Dynamic dispatch implemented with vtables

A pointer to a class C object can be treated as a pointer to a class A object

```cpp
class C *c = ...;
class A *a = c;
a->m2(3);
*(a->vtable[1])(a, 3);
```
**New Topic: Exceptions (informal description)**

If expression e evaluates “normally” to value \(v\), then \(v\) is the result of the entire expression.

Otherwise, an exceptional value \(v'\) is “raised” in the evaluation of e, then result is \((f \ v')\)

Evaluate expression e to value \(v\), and then raise \(v\) as an exceptional value, which can only be “handled”.

Implementation of exceptions may require a lot of language-specific consideration and care. Exceptions can interact in powerful and unexpected ways with other language features. Think of C++ and class destructors, for example.

**Viewed from the call stack**

Call stack just before evaluating code for 
\(e \ handle \ f\)

Push a special frame for the handle

“raise \(v\)” is encountered while evaluating a function body associated with top-most frame

“Unwind” call stack. Depending on language, this may involve some “clean up” to free resources.
**Possible pseudo-code implementation**

```plaintext
let fun _h27 () =
  build special "handle frame"
  save address of f in frame;
  ... code for e ...
  return value of e
in _h27 () end

raise e

... code for e ...
save v, the value of e;
unwind stack until first
fp found pointing at a handle frame;
Replace handle frame with frame
for call to (extracted) f using
v as argument.
```

**New topic: Bootstrapping a compiler**

- Compilers compiling themselves!
- Read Chapter 13 Of
  - Basics of Compiler Design
  - by Torben Mogensen
    http://www.diku.dk/hjemmesider/ansatte/torbenm/Basics/

http://mythologian.net/ouroboros-symbol-of-infinity/

Bootstrapping. We need some notation . . .

**Simple Examples**

- An application called **app** written in language **A**
- An interpreter or VM for language **A**
  - Written in language **B**
- A machine called **mch** running language **A** natively.

**Tombstones**

- This is an application called **trans**
  - That translates programs in language **A** into programs in language **B**, and it is written in language **C**.
Thanks to David Greaves for the example.

Translator `foo.B` is produced as output from `trans` when given `foo.A` as input.
Our seemingly impossible task

We have just invented a really great new language L (in fact we claim that “L is far superior to C++”). To prove how great L is we write a compiler for L in L (of course!). This compiler produces machine code B for a widely used instruction set (say B = x86).

Furthermore, we want to compile our compiler so that it can run on a machine running B. Our compiler is written in L! How can we compiler our compiler?

There are many many ways we could go about this task. The following slides simply sketch out one plausible route to fame and fortune.

Step 1
Write a small interpreter (VM) for a small language of byte codes

MBC = My Byte Codes

The zoom machine!
Step 2
Pick a small subset $S$ of $L$ and write a translator from $S$ to MBC

Write `comp_1.cpp` by hand. (It sure would be nice if we could hide the fact that this is written in C++.)

Compiler `comp_1.B` is produced as output from `gcc` when `comp_1.cpp` is given as input.

Step 3
Write a compiler for $L$ in $S$

Write a compiler `comp_2.S` for the full language $L$, but written only in the sub-language $S$.

Compile `comp_2.S` using `comp_1.B` to produce `comp_2.mbc`
Step 4
Write a compiler for \( L \) in \( L \), and then compile it!

Rewrite/extend compiler \( \text{comp}_2.S \) to produce \( \text{comp}.L \) using the full power of language \( L \).

Putting it all together

We wrote these compilers and the MBC VM.
Step 5: Cover our tracks and leave the world mystified and amazed!

Our L compiler download site contains only three components:

- **MBC**
- **comp_2.mbc**
- **L**
- **comp.L**
- **B**
- **MBC**

**comp_2.mbc** is a just file of **bytes**. We give it the mysterious name such as **mr-e**

Our instructions:
1. Use **gcc** to compile the **zoom** interpreter
2. Use **zoom** to run **mr-e** with input **comp.L** to output the compiler **comp.B**. MAGIC!

---

Another example (Mogensen, Page 285)

Solving a different problem.

You have:
1. An ML compiler on ARM. Who knows where it came from.
2. An ML compiler written in ML, generating x86 code.

You want:
An ML compiler generating x86 and running on an x86 platform.