CS 516: COMPILERS

Lecture 6

Topics

- Intermediate Representations

Materials

- lec06.zip (ir1, ir2, ir3, …)

INTERMEDIATE REPRESENTATIONS

Eliminating Nested Expressions

- Fundamental problem:
  - Compiling complex & nested expression forms to simple operations.

Source

```
((1 + X4) + (3 + (X1 * 5)))
```

AST

```
Add(Add(Const 1, Var X4),
    Add(Const 3, Mul(Var X1,
                     Const 5)))
```

IR

?

- Idea: *name* intermediate values, make order of evaluation explicit.
  - No nested operations.

Translation to SLL

- Given this:

```
Add(Add(Const 1, Var X4),
    Add(Const 3, Mul(Var X1,
                     Const 5)))
```

- Translate to this desired SLL form:

```
let tmp0 = add 1L varX4 in
let tmp1 = mul varX1 5L in
let tmp2 = add 3L tmp1 in
let tmp3 = add tmp0 tmp2 in
  tmp3
```

- Translation makes the order of evaluation explicit.
- Names intermediate values
- Note: introduced temporaries are *never modified*

SIMPLE LET-BASED IR

Intermediate Representations

- IR1: Expressions
  - simple arithmetic expressions, immutable global variables

- IR2: Commands
  - global *mutable* variables
  - commands for update and sequencing

- IR3: Local control flow
  - conditional commands & while loops
  - *basic blocks*

- IR4: Procedures (top-level functions)
  - local state
  - call stack

IR1

Source: Arith. Expressions

```
type exp =
  | Var of var
  | Const of int64
  | Add of exp * exp
  | Mul of exp * exp
  | Neg of exp
```

IR1: "let" instructions

```
module IR = struct
  (* Unique identifiers for temporaries. *)
  type uid = int

  (* "gensym" -- generate a new unique identifier *)
  let mk_uid : unit -> uid =
    let ctr = ref 0 in
    fun () -> let uid = !ctr in
      ctr := !ctr + 1;
      uid

  (* syntactic values *)
  type opn =
    | Id of uid
    | Const of int64
    | Var of var

  (* binary operations *)
  type bop =
    | Add
    | Mul

  (* instructions *)
  (* note that there is no nesting of operations! *)
  type insn =
    | Let of uid * bop * opn * opn
    (* e.g.  "let tmp0 = add 1L varX4 in" *)

  type program = {
    insns: insn list;
    ret: opn
  }
```

IR1

Source: Arith. Expressions

```
type exp =
  | Var of var
```

IR1: "let" instructions

```
module IR = struct
  (* Unique identifiers for temporaries. *)
```

```
let program : int64 =
  (1L +. varX4) +. (3L +. (varX1 *. 5L))
```

translate

```
let program : int64 =
```

```
  type program = {
    insns: insn list;
    ret: opn
  }
```

IR1

Source: Arith. Expressions

```
type exp =
```

IR1: "let" instructions

```
module IR = struct
  (* Unique identifiers for temporaries. *)
```

```
let program : int64 =
  (1L +. varX4) +. (3L +. (varX1 *. 5L))
```

translate

```
let program : int64 =
  let tmp1 = add 1L varX4 in
  let tmp2 = mul varX1 5L in
  let tmp3 = add 3L tmp2 in
  let tmp4 = add tmp1 tmp3 in
  ret tmp4
```

```
type program = {
  insns: insn list;
  ret: opn
}
```

IR1

Source: Arith. Expressions

```
type exp =
    | Var of var
    | Const of int64
    | Add of exp * exp
    | Mul of exp * exp
    | Neg of exp
```

IR1: "let" instructions

```
module IR = struct
  (* Unique identifiers for temporaries. *)
  type uid = int

  (* "gensym" -- generate a new unique identifier *)
  let mk_uid : unit -> uid =
    let ctr = ref 0 in
    fun () -> let uid = !ctr in
      ctr := !ctr + 1;
      uid

  (* syntactic values *)

                                  t64

  (* binary operations *)

              ons *)
  (* note that there is no nesting of operations! *)
  type insn =
    | Let of uid * bop * opn * opn
    (* e.g.  "let tmp0 = add 1L varX4 in" *)

  type program = {
    insns: insn list;
    ret: opn
  }
```

IR1

Source: Arith. Expressions

IR1: "let" instructions

```
type exp =
```

```
module IR = struct
```

```
module Compile = struct
  open SRC

  (* Expressions produce answers, so the result of compiling an expression
     is a list of instructions and an operand that will contain the final
     result of comping the expression.

     - we can share the code common to binary operations.
  *)

  let rec compile_exp (e:exp) : (IR.insn list) * IR.opn =
    let compile_bop bop e1 e2 =




    in
    begin match e with
      |
      |
      |
      |
      |
    end

  let compile (e:exp) : IR.program =
    let insns, ret = compile_exp e in
    IR.{ insns; ret }

end
```

```
temporaries. *)
new unique identifier *)
=
r in
nesting of operations! *)
n * opn
d 1L varX4 in" *)
    insns: insn list;
    ret: opn
  }
```

IR1

Source: Arith. Expressions

```
type exp =
```

IR1: “let” instructions

```
module Compile = struct
  open SRC

  (* Expressions produce answers, so the result of compiling an expression
     is a list of instructions and an operand that will contain the final
     result of comping the expression.

     - we can share the code common to binary operations.
  *)

  let rec compile_exp (e:exp) : (IR.insn list) * IR.opn =
    let compile_bop bop e1 e2 =




    in
    begin match e with
      | Var x         -> [], IR.Var x
      | Const c       -> [], IR.Const c
      | Add(e1, e2) -> compile_bop IR.Add e1 e2
      | Mul(e1, e2) -> compile_bop IR.Mul e1 e2
      | Neg(e1)       -> compile_bop IR.Mul e1 (Const(-1L))
    end

  let compile (e:exp) : IR.program =
    let insns, ret = compile_exp e in
    IR.{ insns; ret }

end
```

```
module IR = struct
 temporaries. *)
 new unique identifier *)
=
r in
nesting of operations! *)
n * opn
d 1L varX4 in" *)
    insns: insn list;
    ret: opn
  }
```

IR1

Source: Arith. Expressions

IR1: "let" instructions

```
type exp =
```

```
module Compile = struct
  open SRC

  (* Expressions produce answers, so the result of compiling an expression
     is a list of instructions and an operand that will contain the final
     result of comping the expression.

     - we can share the code common to binary operations.
  *)

  let rec compile_exp (e:exp) : (IR.insn list) * IR.opn =
    let compile_bop bop e1 e2 =
        let ins1, ret1 = compile_exp e1 in
        let ins2, ret2 = compile_exp e2 in
        let ret = IR.mk_uid () in
        ins1 @ ins2 @ IR.[Let (ret, bop, ret1, ret2)], IR.Id ret
    in
    begin match e with
      | Var x         -> [], IR.Var x
      | Const c       -> [], IR.Const c
      | Add(e1, e2) -> compile_bop IR.Add e1 e2
      | Mul(e1, e2) -> compile_bop IR.Mul e1 e2
      | Neg(e1)       -> compile_bop IR.Mul e1 (Const(-1L))
    end

  let compile (e:exp) : IR.program =
    let insns, ret = compile_exp e in
    IR.{ insns; ret }

end
```

```
module IR = struct
 temporaries. *)
 new unique identifier *)
=
r in
nesting of operations! *)
n * opn
d 1L varX4 in" *)
    insns: insn list;
    ret: opn
  }
```

code demo

IR1

1. **Developing an IR. Step 1: Arithmetic Expressions.**

```
unzip lec06.zip ; cd lec06/ ; make
```

   A. Look at the Makefile
   B. **`code/ir1.ml`**
   C. etc.

IR2

Source: Now with commands and mutable global variables.

IR2: Now with `load` and `store`

```
(* Abstract syntax of arithmetic expressions *)
type exp =
  | Var of var
  | Add of exp * exp
  | Mul of exp * exp
  | Neg of exp
  | Const of int64

(* Abstract syntax of commands *)
type cmd =
  | Skip                          (* skip     *)
  | Assn of var * exp             (* X := e   *)
  | Seq  of cmd * cmd             (* c1 ; c2 *)
```

IR2

Source: Now with commands and mutable global variables.

```
(* Abstract syntax of arith
type exp =
  | Var of var
  | Add of exp * exp
  | Mul of exp * exp
  | Neg of exp
  | Const of int64

(* Abstract syntax of comma
type cmd =
  | Skip
  | Assn of var * exp
  | Seq  of cmd * cmd
```

IR2: Now with `load` and `store`

```
(* operands *)
type opn =
  | Id of uid
  | Const of int64

(* binary operations *)
type bop =
  | Add
  | Mul

(* instructions *)
(* note that there is no nesting of ope
type insn =
  | Let of uid * bop * opn * opn
  | Load of uid * var
  | Store of var * opn

type program = {
  insns: insn list
}
```

IR3

Source: Now with if/while

```
(* Abstract syntax of arithmetic ex
type exp =
  | Var of var
  | Add of exp * exp
  | Mul of exp * exp
  | Neg of exp
  | Const of int64

(* Abstract syntax of commands *)
type cmd =
  | Skip
  | Assn of var * exp
  | Seq  of cmd * cmd
  | IfNZ of exp * cmd * cmd
  | WhileNZ of exp * cmd
```

IR3: Control flow graphs

Basic Blocks (IR3)

- A sequence of instructions that is always executed starting at the first instruction and always exits at the last instruction.
  - Starts with a label that names the *entry point* of the basic block.
  - Ends with a control-flow instruction (e.g. branch or return) the "link"
  - Contains no other control-flow instructions
  - Contains no interior label used as a jump target

- Basic blocks can be arranged into a *control-flow graph*
  - Nodes are basic blocks
  - There is a directed edge from node A to node B if the control flow instruction at the end of basic block A might jump to the label of basic block B.

IR3

Source: Now with if/while

```
X2 := X1 + X2;
IFNZ X2 THEN {
  X1 := X1 + 1
} ELSE {
  X2 := X1
} ;
X2 := X2 * X1
```

IR3: Control flow graphs

entry:

```
let tmp1 = load X1 in
let tmp2 = load X2 in
let tmp3 = add tmp1 tmp2 in
let _ = store tmp3 X2 in
let tmp4 = load x2 in
let tmp5 = icmp eq tmp 0L in
cbr tmp5 branch1 branch2
```

branch1:

```
let tmp5 = load X1 in
let tmp6 = add tmp5 1L in
let _ = store tmp6 X1 in
br merge
```

branch2:

```
let tmp7 = load X1 in
let _ = store tmp 7 X2 in
br merge
```

merge:

```
let tmp8 = load X2 in
let tmp9 = load X1 in
let tmp10 = mul tmp8 tmp9 in
let _ = store tmp10 X2 in
ret ()
```

IR3

Source: Now with if/while

```
(* Abstract syntax of arithmeti
type exp =
  | Var of var
  | Add of exp * exp
  | Mul of exp * exp
  | Neg of exp
  | Const of int64

(* Abstract syntax of commands *
type cmd =
  | Skip
  | Assn of var * exp
  | Seq  of cmd * cmd
  | IfNZ of exp * cmd * cmd
  | WhileNZ of exp * cmd
```

IR3: Control flow graphs

```
(* operands *)
type opn =
  | Id of uid
  | Const of int64

(* binary arithmetic operations *)
type bop =
  | Add
  | Mul

(* comparison operations *)
type cmpop =
  | Eq
  | Lt

(* instructions *)
(* note that there is no nesting of operations! *)
type insn =
  | Let of uid * bop * opn * opn
  | Load of uid * var
  | Store of var * opn
```

IR3

Source: Now with if/while

```
(* Abstract syntax of arithmeti
type exp =
    | Var of var
    | Add of exp * exp
    | Mul of exp * exp
    | Neg of exp
    | Const of int64

(* Abstract syntax of commands *
type cmd =
    | Skip
    | Assn of var * exp
    | Seq  of cmd * cmd
    | IfNZ of exp * cmd * cmd
    | WhileNZ of exp * cmd
```

IR3: Control flow graphs

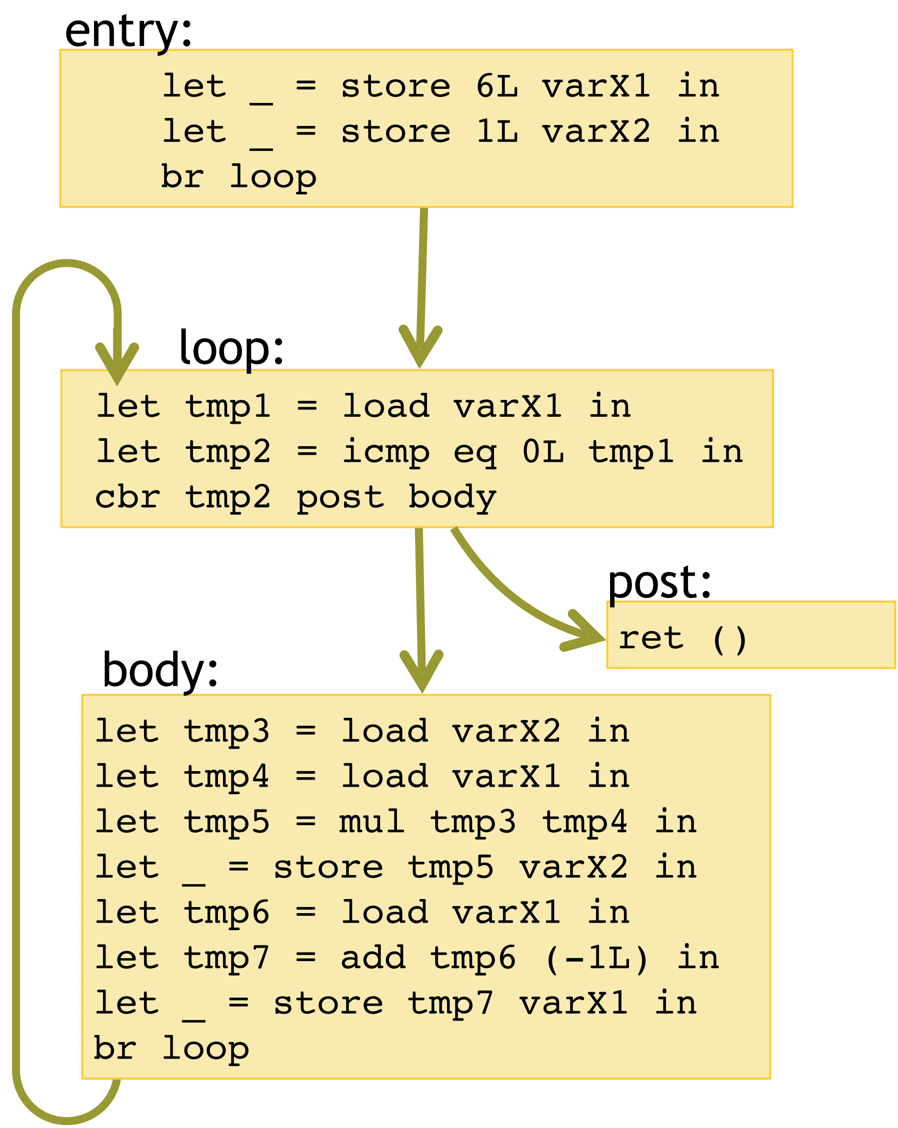
```
(* operands *)
type opn =
    | Id of uid
    | Const of int64

(* binary arithmetic operations *)
type bop =
    | Add
    | Mul

(* comparison operations *)
type cmpop =
    | Eq
    | Lt

(* instructions *)
(* note that there is no nesting of operations! *)
type insn =
    | Let of uid * bop * opn * opn
    | Load of uid * var
    | Store of var * opn
    | ICmp of uid * cmpop * opn * opn

type terminator =
    | Ret
    | Br of lbl                  (* unconditional branch *)
    | Cbr of opn * lbl * lbl     (* conditional branch *)

(* Basic blocks *)
type block = { insns: insn list; terminator: terminator }

(* Control Flow Graph: a pair of an entry block and a set
type cfg = block * (lbl * block) list
```

IR3

Source: Now with if/while

```
X1 := 6;
X2 := 1;
WhileNZ X1 DO
  X2 := X2 * X1;
  X1 := X1 + (-1);
DONE
```

IR3: Control flow graphs



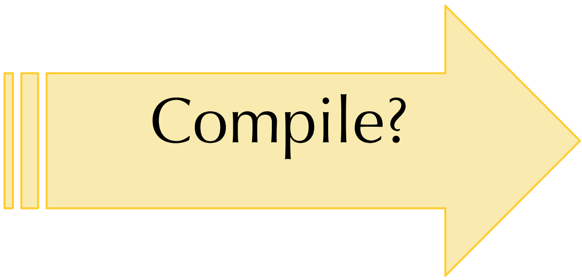

```
type insn =
  | Let of uid * bop * opn * opn
  | Load of uid * var
  | Store of var * opn
  | ICmp of uid * cmpop * opn * opn

type terminator =
  | Ret
  | Br of lbl              (* unconditional branch *)
  | Cbr of opn * lbl * lbl (* conditional branch *)

(* Basic blocks *)
type block = { insns: insn list; terminator: terminator }

(* Control Flow Graph: a pair of an entry block and a set
type cfg = block * (lbl * block) list
```

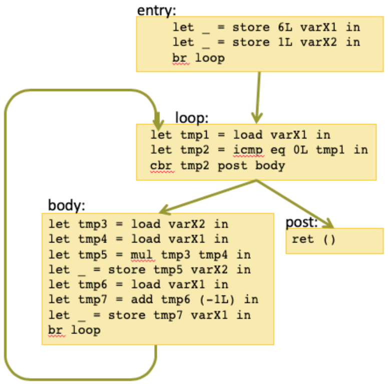
IR3

Source: Now with if/while

```
WhileNZ X1 DO
  X2 := X2 * X1;
  X1 := X1 + (-1);
DONE
```

IR3: Control flow graphs





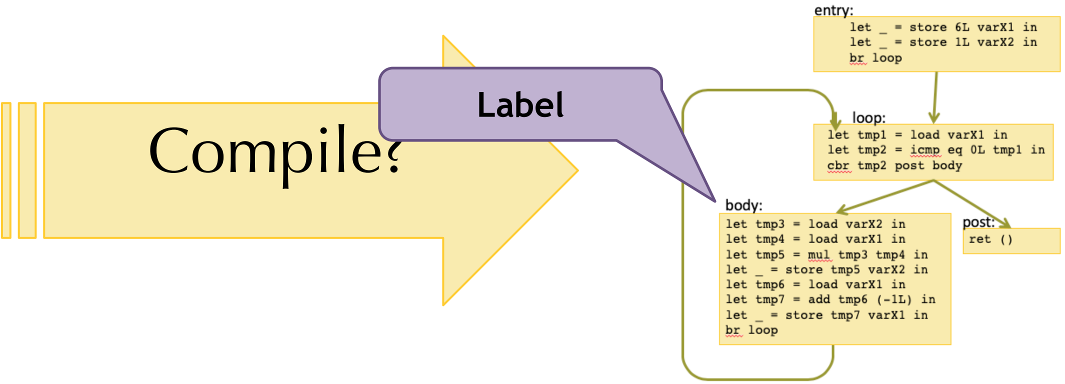
Compilation emits (instructions, operand), not graphs.

IR3

Source: Now with if/while

```
WhileNZ X1 DO
   X2 := X2 * X1;
   X1 := X1 + (-1);
DONE
```

IR3: Control flow graphs



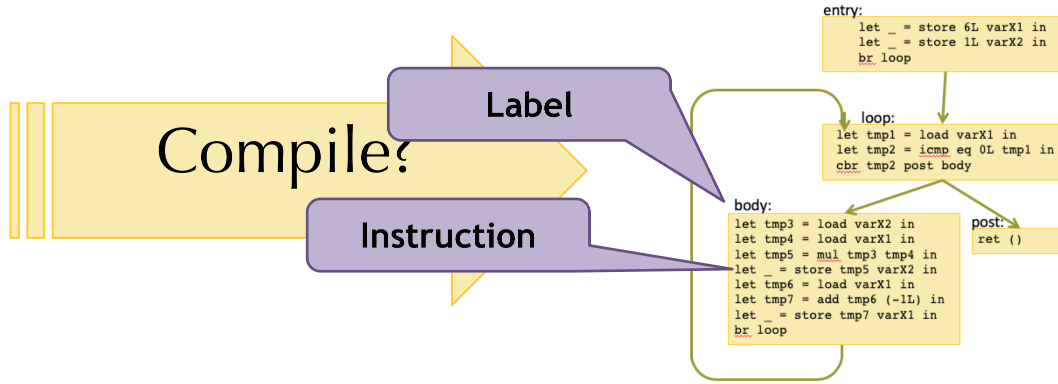
Compilation emits (instructions, operand), not graphs.

IR3

Source: Now with if/while

```
WhileNZ X1 DO
  X2 := X2 * X1;
  X1 := X1 + (-1);
DONE
```

IR3: Control flow graphs



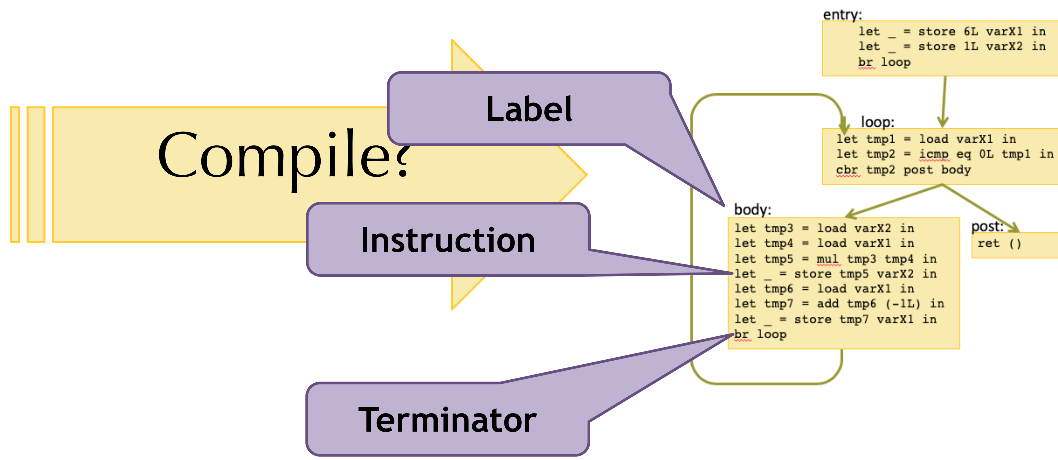
Compilation emits (instructions, operand), not graphs.

IR3

Source: Now with if/while

```
WhileNZ X1 DO
   X2 := X2 * X1;
   X1 := X1 + (-1);
DONE
```

IR3: Control flow graphs



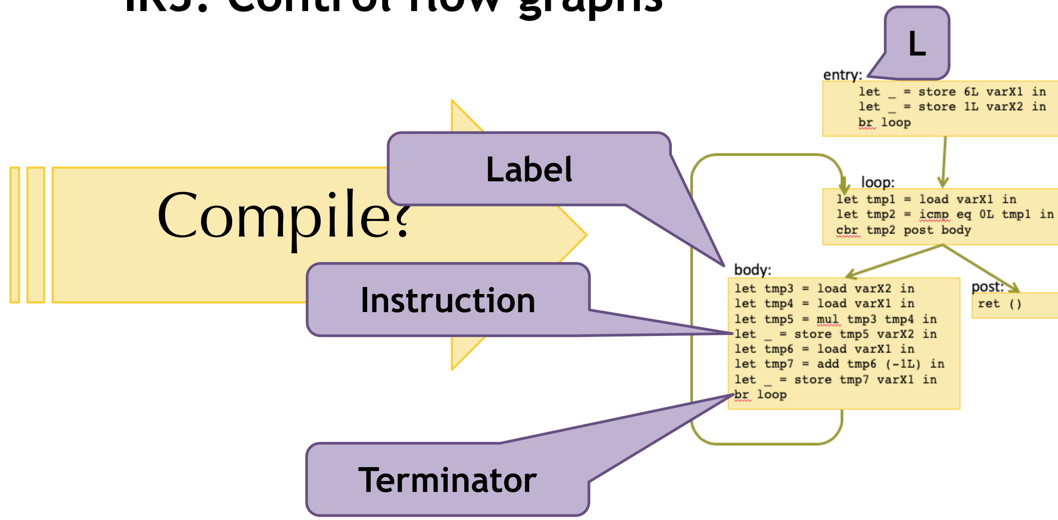
Compilation emits (instructions, operand), not graphs.

IR3

Source: Now with if/while

```
WhileNZ X1 DO
   X2 := X2 * X1;
   X1 := X1 + (-1);
DONE
```

IR3: Control flow graphs



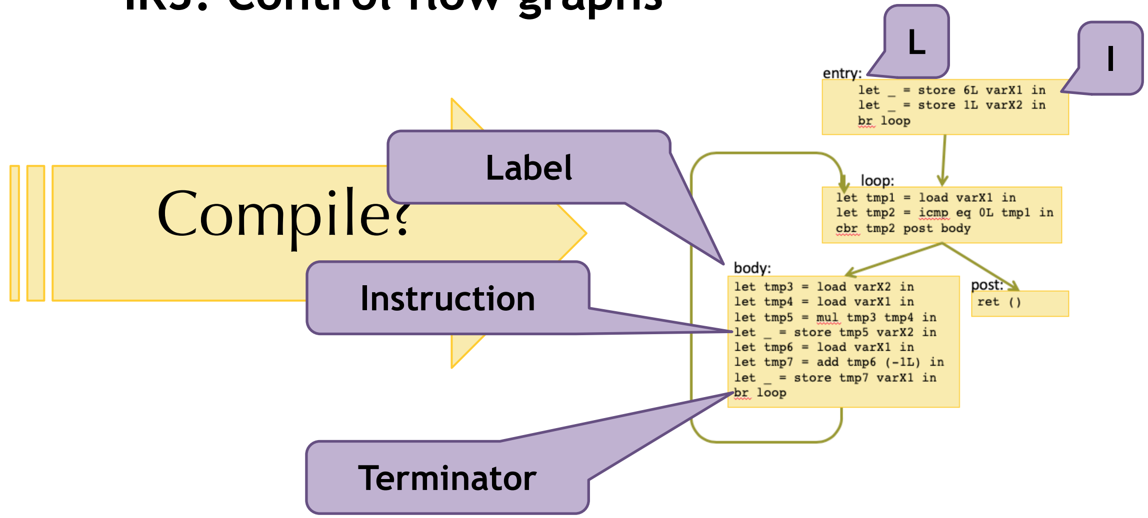
Compilation emits (instructions, operand), not graphs.

IR3

Source: Now with if/while

```
WhileNZ X1 DO
  X2 := X2 * X1;
  X1 := X1 + (-1);
DONE
```

IR3: Control flow graphs



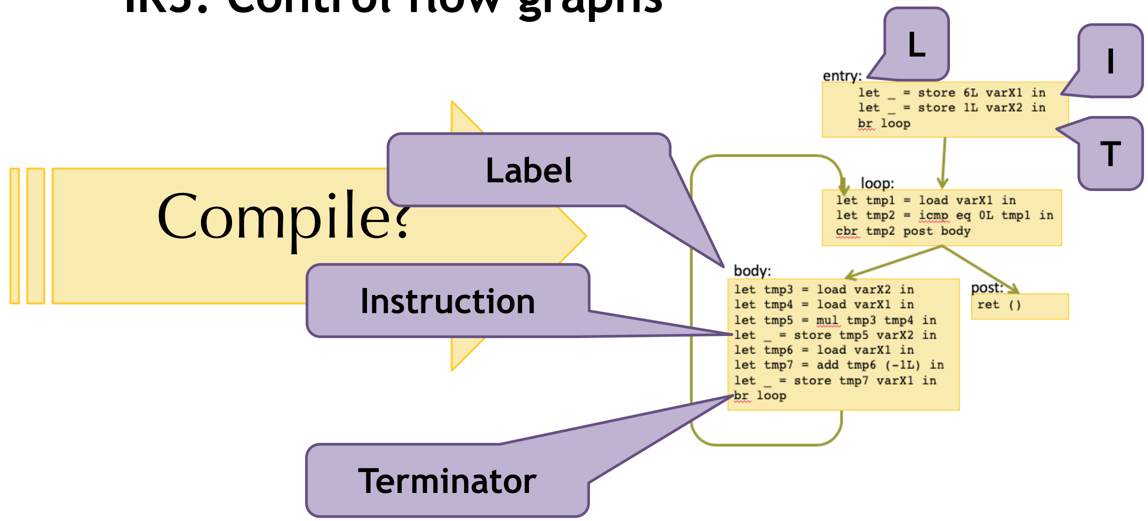
Compilation emits (instructions, operand), not graphs.

IR3

Source: Now with if/while

```
WhileNZ X1 DO
   X2 := X2 * X1;
   X1 := X1 + (-1);
DONE
```

IR3: Control flow graphs



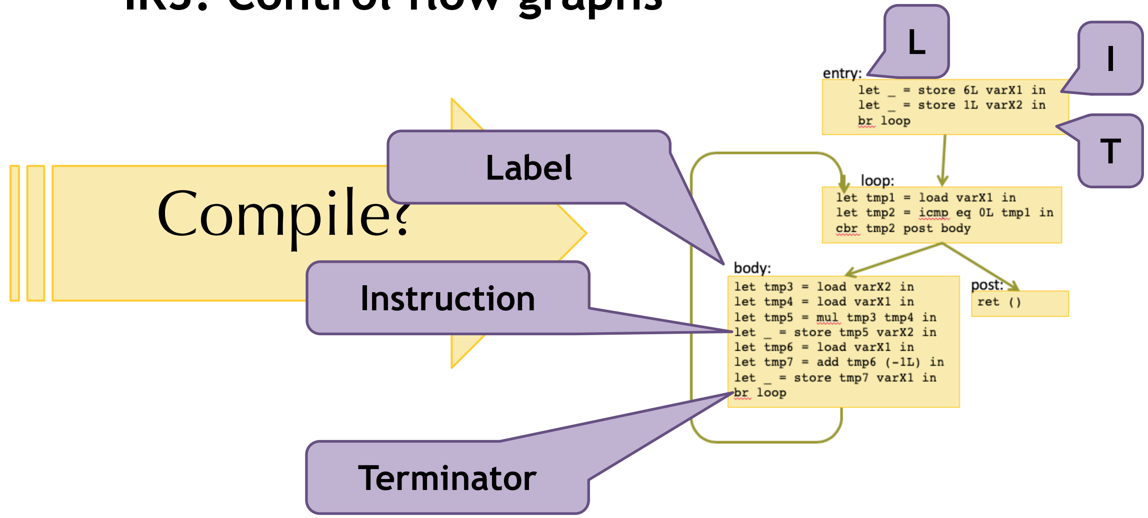
Compilation emits (instructions, operand), not graphs.

IR3

Source: Now with if/while

```
WhileNZ X1 DO
  X2 := X2 * X1;
  X1 := X1 + (-1);
DONE
```

IR3: Control flow graphs



Compilation emits (instructions, operand), not graphs.

Idea: Compilation emits lists of tagged instructions

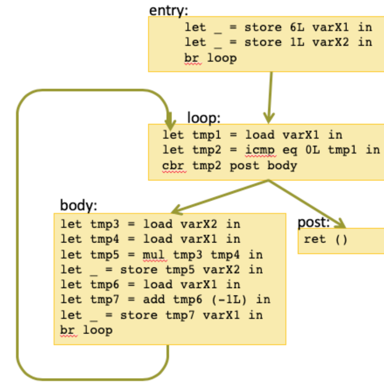
where tag ∈ {Label, Instruction, Terminator}

IR3

Source: Now with if/while

```
WhileNZ X1 DO
   X2 := X2 * X1;
   X1 := X1 + (-1);
DONE
```

IR3: Control flow graphs



Create stream

Stream

L	"loop1"
I	let tmp1 = load varX1
I	let tmp2 = icmp eq 0L tmp1
T	cbr tmp2 "post1" "body1"
L	"body1"
I	let tmp3 = load varX1
I	…
T	br "loop1"
L	"post"
…	…

Construct graph (build_cfg)

IR3

- See ir3.ml in lec06.zip

IR3

```
(* Convert an instruction stream into a control flow graph.
   - assumes that the instructions are in 'reverse' order of execution.
*)
let build_cfg (code:stream) : cfg   =
  let blocks_of_stream (code:stream) =
    let (insns, term_opt, blks) =  List.fold_left
        (fun (insns, term_opt, blks) e ->
          begin match e with
            | L l ->
              begin match term_opt with
                | None ->
                  if (List.length insns) = 0 then  ([], None, blks)
                  else failwith @@
                    Printf.sprintf "build_cfg: block labeled %s has\
                                    no terminator" l

                | Some terminator ->
                  ([], None, (l, {insns; terminator})::blks)
              end
            | T t  -> ([], Some t, blks)
            | I i  -> (i::insns, term_opt, blks)
          end)
        ([], None, []) code
    in
    begin match term_opt with
      | None -> failwith "build_cfg: entry block has no terminator"
      | Some terminator ->
        ({insns; terminator}, blks)
    end
  in
  blocks_of_stream code
```

IR3

```
(* Convert an instruction stream into a control flow graph.
   - assumes that the instructions are in 'reverse' order of execution.
*)
let build_cfg (code:stream) : cfg   =
  let blocks_of_stream (code:stream) =
    let (insns, term_opt, blks) =  List.fold_left
        (fun (insns, term_opt, blks) e ->
          begin match e with
            | L l ->
              begin match term_opt with
                | None ->
                  if (List.length insns) = 0 then  ([], None, blks)
                  else failwith @@
                    Printf.sprintf "build_cfg: block labeled %s has\
                                    no terminator" l

                | Some terminator ->
                  ([], None, (l, {insns; terminator})::blks)
              end
            | T t  -> ([], Some t, blks)
            | I i  -> (i::insns, term_opt, blks)
          end)
        ([], None, []) code
    in
    begin match term_opt with
      | None -> failwith "build_cfg: entry block has no terminator"
      | Some terminator ->
        ({insns; terminator}, blks)
    end
  in
  blocks_of_stream code
```

IR3

Accumulations

```
(* Convert an instruction stream into a control flow graph.
   - assumes that the instructions are in 'reverse' order of execution.
*)
let build_cfg (code:stream) : cfg   =
  let blocks_of_stream (code:stream) =
    let (insns, term_opt, blks) =  List.fold_left
        (fun (insns, term_opt, blks) e ->
          begin match e with
            | L l ->
              begin match term_opt with
                | None ->
                  if (List.length insns) = 0 then  ([], None, blks)
                  else failwith @@
                    Printf.sprintf "build_cfg: block labeled %s has\
                                    no terminator" l

                | Some terminator ->
                  ([], None, (l, {insns; terminator})::blks)
              end
            | T t  -> ([], Some t, blks)
            | I i  -> (i::insns, term_opt, blks)
          end)
        ([], None, []) code
    in
    begin match term_opt with
      | None -> failwith "build_cfg: entry block has no terminator"
      | Some terminator ->
        ({insns; terminator}, blks)
    end
  in
  blocks_of_stream code
```

IR3

Accumulations

Each element

```
(* Convert an instruction stream into a control flow graph.
   - assumes that the instructions are in 'reverse' order of execution.
*)
let build_cfg (code:stream) : cfg  =
  let blocks_of_stream (code:stream) =
    let (insns, term_opt, blks) =  List.fold_left
      (fun (insns, term_opt, blks) e ->
        begin match e with
          | L l ->
            begin match term_opt with
              | None ->
                if (List.length insns) = 0 then  ([], None, blks)
                else failwith @@
                  Printf.sprintf "build_cfg: block labeled %s has\
                                  no terminator" l

              | Some terminator ->
                ([], None, (l, {insns; terminator})::blks)
            end
          | T t  -> ([], Some t, blks)
          | I i  -> (i::insns, term_opt, blks)
        end)
      ([], None, []) code
    in
    begin match term_opt with
      | None -> failwith "build_cfg: entry block has no terminator"
      | Some terminator ->
        ({insns; terminator}, blks)
    end
  in
  blocks_of_stream code
```

IR3

```
(* Convert an instruction stream into a control flow graph.
   - assumes that the instructions are in 'reverse' order of execution.
*)
let build_cfg (code:stream) : cfg  =
  let blocks_of_stream (code:stream) =
    let (insns, term_opt, blks) =  List.fold_left
      (fun (insns, term_opt, blks) e ->
        begin match e with
          | L l ->
            begin match term_opt with
              | None ->
                if (List.length insns) = 0 then  ([], None, blks)
                else failwith @@
                  Printf.sprintf "build_cfg: block labeled %s has\
                                  no terminator" l

              | Some terminator ->
                ([], None, (l, {insns; terminator})::blks)
            end
          | T t  -> ([], Some t, blks)
          | I i  -> (i::insns, term_opt, blks)
        end)
      ([], None, []) code
    in
    begin match term_opt with
      | None -> failwith "build_cfg: entry block has no terminator"
      | Some terminator ->
        ({insns; terminator}, blks)
    end
  in
  blocks_of_stream code
```

Accumulations

Each element

Working backward, new terminator and empty instructions

IR3

```
(* Convert an instruction stream into a control flow graph.
   - assumes that the instructions are in 'reverse' order of execution.
*)
let build_cfg (code:stream) : cfg   =
  let blocks_of_stream (code:stream) =
    let (insns, term_opt, blks) =  List.fold_left
        (fun (insns, term_opt, blks) e ->
          begin match e with
            | L l ->
              begin match term_opt with
                | None ->
                  if (List.length insns) = 0 then  ([], None, blks)
                  else failwith @@
                    Printf.sprintf "build_cfg: block labeled %s has\
                                    no terminator" l

                | Some terminator ->
                  ([], None, (l, {insns; terminator})::blks)
              end
            | T t  -> ([], Some t, blks)
            | I i  -> (i::insns, term_opt, blks)
          end)
        ([], None, []) code
    in
    begin match term_opt with
      | None -> failwith "build_cfg: entry block has no terminator"
      | Some terminator ->
        ({insns; terminator}, blks)
    end
  in
  blocks_of_stream code
```

Accumulations

Each element

Working backward, new terminator and empty instructions

Just accumulate the instruction

IR3

```
(* Convert an instruction stream into a control flow graph.
 - assumes that the instructions are in 'reverse' order of execution.
*)
let build_cfg (code:stream) : cfg  =
  let blocks_of_stream (code:stream) =
    let (insns, term_opt, blks) =  List.fold_left
      (fun (insns, term_opt, blks) e ->
        begin match e with
          | L l ->
            begin match term_opt with
              | None ->
                if (List.length insns) = 0 then  ([], None, blks)
                else failwith @@
                  Printf.sprintf "build_cfg: block labeled %s has\
                                          no terminator" l

              | Some terminator ->
                ([], None, (l, {insns; terminator})::blks)
            end
          | T t  -> ([], Some t, blks)
          | I i  -> (i::insns, term_opt, blks)
        end)
      ([], None, []) code
    in
    begin match term_opt with
      | None -> failwith "build_cfg: entry block has no terminator"
      | Some terminator ->
        ({insns; terminator}, blks)
    end
  in
  blocks_of_stream code
```

Accumulations

Each element

Now construct the block from the accumulated instructions and terminator

Working backward, new terminator and empty instructions

Just accumulate the instruction

IR3

```
(* Convert an instruction stream into a control flow graph.
   - assumes that the instructions are in 'reverse' order of execution.
*)
let build_cfg (code:stream) : cfg   =
  let blocks_of_stream (code:stream) =
    let (insns, term_opt, blks) =  List.fold_left
      (fun (insns, term_opt, blks) e ->
        begin match e with
          | L l ->
            begin match term_opt with
              | None ->
                if (List.length insns) = 0 then  ([], None, blks)
                else failwith @@
                  Printf.sprintf "build_cfg: block labeled %s has\
                                  no terminator" l

              | Some terminator ->
                ([], None, (l, {insns; terminator})::blks)
            end
          | T t  -> ([], Some t, blks)
          | I i  -> (i::insns, term_opt, blks)
        end)
      ([], None, []) code
    in
    begin match term_opt with
      | None -> failwith "build_cfg: entry block has no terminator"
      | Some terminator ->
        ({insns; terminator}, blks)
    end
  in
  blocks_of_stream code
```

Accumulations

Each element

Now construct the block from the accumulated instructions and terminator

Working backward, new terminator and empty instructions

Just accumulate the instruction

What remains is an unlabeled (but terminated) block

IR4

Source Code

```
int64 square(int64 x) {
  x = x + 1;
  return (x * x);
}

void caller() {
  int x = 3;
  int y = square(x);
  print ( y + x );
}
```

IR4: Top-level Functions & Stack variables

```
(* instructions *)
(* note that there is no nesting of operations! *)
type insn =
  | Let of uid * bop * opn * opn
  | Load of uid * var
  | Store of var * opn
  | ICmp of uid * cmpop * opn * opn

type terminator =
  | Ret
  | Br of lbl                    (* unconditional branch *)
  | Cbr of opn * lbl * lbl       (* conditional branch *)

(* Basic blocks *)
type block = { insns: insn list; terminator: terminator }

(* Control Flow Graph: a pair of an entry block and a set labe
type cfg = block * (lbl * block) list

type program = {
  fdecls : fdecl list
}
```

IR4

Source Code

```
int64 square(int64 x) {
  x = x + 1;
  return (x * x);
}

void caller() {
  int x = 3;
  int y = square(x);
  print ( y + x );
}
```

IR4: Top-level Functions & Stack variables

```
(* instructions *)
(* note that there is no nesting of operations! *)
type insn =
  | Let of uid * bop * opn * opn
  | Load of uid * var
  | Store of var * opn
  | ICmp of uid * cmpop * opn * opn
  | Call of uid * fn_name * (opn list)
  | Alloca of uid

type terminator =
  | Ret
  | Br of lbl                (* unconditional branch *)
  | Cbr of opn * lbl * lbl   (* conditional branch *)

(* Basic blocks *)
type block = { insns: insn list; terminator: terminator }

(* Control Flow Graph: a pair of an entry block and a set labe
type cfg = block * (lbl * block) list

type fdecl = { name: fn_name;  param : uid list; cfg : cfg }

type program = {
  fdecls : fdecl list
}
```

IR is independent of calling convention

IR5

Source Code

```
int64 square(int64 x) {
  x = x + 1;
  return (x * x);
}

void caller() {
  int x = 3;
  int y = square(x);
  print ( y + x );
}
```

IR5: Unify vars and fn_name into global identifiers

```
type uid = string
type lbl = string
type gid = string
```

```
(* instructions *)
(* note that there is no nesting of operations! *)
(* pull out the common 'uid' element from these constructors *)
type insn =
  | Binop of bop * opn * opn     (* Rename let to binop *)
  | Load of gid
  | Store of gid * opn
  | ICmp of cmpop * opn * opn
  | Call of gid * (opn list)
  | Alloca
```