

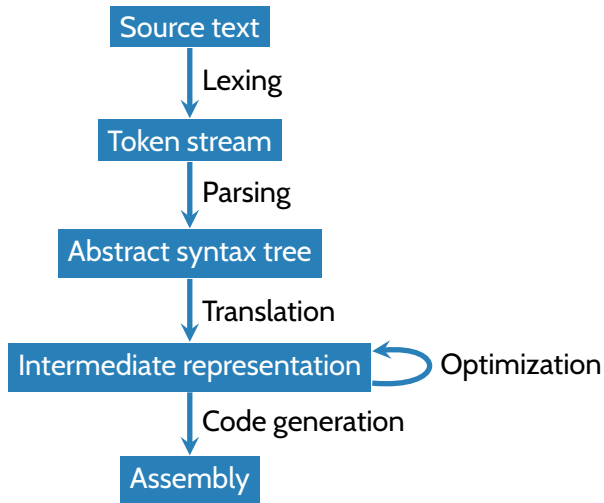
COS320: Compiling Techniques

Zak Kincaid

January 29, 2026

- Reminder: HW1 due **Monday Feb 12**
- Bonus OCaml office hours 4pm Friday Feb 9, in CS 003

Compiler phases (simplified)



Last time: let-based IR

Each instruction has at most three operands (“three-address code”)

`<instr> := let <uid> = <operand> <op> <operand>;`

`| load <uid> = <var>;`

`| store <var> = <operand>;`

`| return <operand>;`

`<operand> := <uid> | <integer>`

`<op> := + | *`

Instructions

Operands

Operations

Control Flow

Concrete syntax

`<instr> ::= let <uid> = <operand> <op> <operand>;
 | load <uid> = <var>;
 | store <var> = <operand>;`

Instructions

`<operand> ::= <uid> | <integer>`

Operands

`<op> ::= + | *`

Operations

`<terminator> ::= br <label>`

Branch

`| cbr <cc> <operand> <label> <label>`

Conditional branch

`| return <operand>`

Return

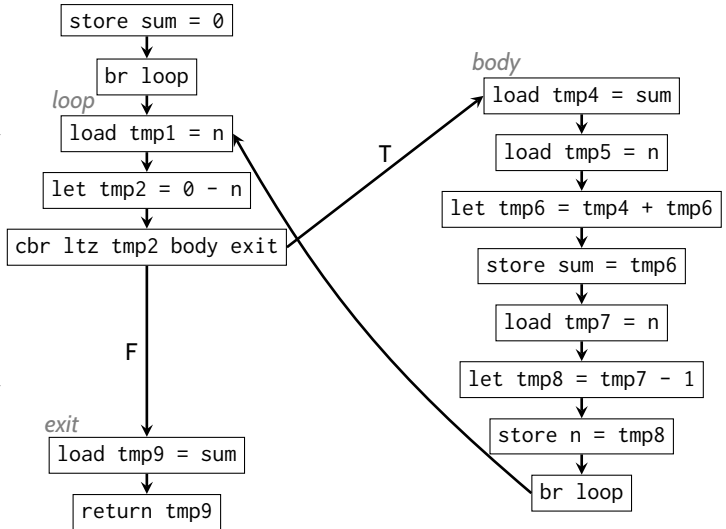
`<cc> ::= EqZ | LeZ | LtZ`

`<block> ::= <instr><block> | <terminator>`

`<program> ::= <program><label>: <block> | <block>`

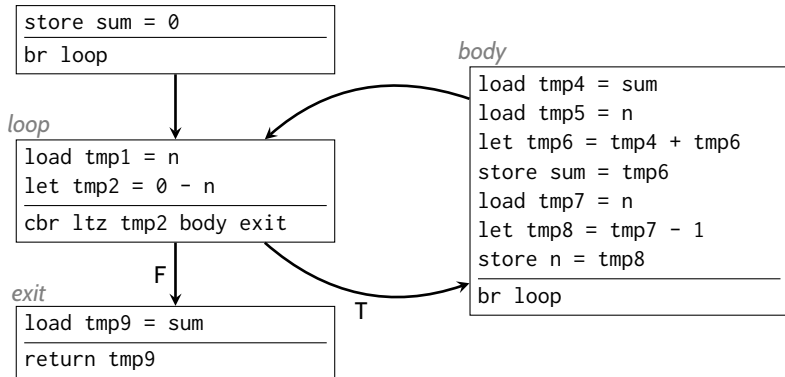
Control Flow Graphs (CFG)

```
int sum_upto(int n) {  
    int sum = 0;  
    while (n > 0) {  
        sum += n;  
        n--;  
    }  
    return sum;  
}
```



Control Flow Graphs (CFG)

```
int sum_upto(int n) {  
  int sum = 0;  
  while (n > 0) {  
    sum += n;  
    n--;  
  }  
  return sum;  
}
```



- Control flow graphs are a graphical representation of the control flow through a procedure
- A *basic block* is a sequence of instructions that
 - ① Starts with an *entry*, which is named by a label
 - ② Ends with a control-flow instruction (br, cbr, or return)
 - the *terminator* of the basic block
 - ③ Contains no interior labels or control flow instructions
- A *control flow graph* (CFG) for a procedure P is a directed, rooted graph where
 - The nodes are basic blocks of P
 - There is an edge $BB_i \rightarrow BB_j$ iff BB_j may execute immediately after BB_i
 - There is a distinguished entry block where the execution of the procedure begins, which has no incoming edges

- CFG models all program executions
 - Every execution corresponds to a path in the CFG, starting at entry
 - Path = sequence of basic blocks B_1, \dots, B_n such that for each i , there is an edge from B_i to B_{i+1}
 - *Simple* path = path without repeated basic blocks
 - (But not vice-versa!)

- CFG models all program executions
 - Every execution corresponds to a path in the CFG, starting at entry
 - Path = sequence of basic blocks B_1, \dots, B_n such that for each i , there is an edge from B_i to B_{i+1}
 - *Simple* path = path without repeated basic blocks
 - (But not vice-versa!)
- Graph structure used extensively in optimization (data flow analysis, loop recognition, ...)
- Simple application: **dead code elimination**
 - 1 Depth-first traversal of the CFG
 - 2 Any *unvisited node* is removed

Why basic blocks?

- Control flow graphs may be defined at the instruction-level rather than basic-block level
- However, there are good reasons for using basic blocks
 - More compact
 - Some optimization passes (“local” optimizations) operate @ basic block level
 - E.g., the implementation of redundant load elimination in `let3.ml`

Constructing a CFG

- “Forwards” algorithm:
 - Traverse statements in IR from top to bottom
 - Find *leaders*: first statement & first statement following a label
 - Basic block = leader up to (but not including) next leader
- Alternately, traverse IR from bottom to top, starting a new basic blocks for each terminator and finishing at label (`build_cfg` in `let3.ml`)
 - (Assumes every label has a corresponding terminator. Does not assume every terminator has a corresponding label—implicitly eliminated dead code)
- Can also construct CFG directly from AST

Generating code from a CFG

- Simple strategy: terminator always compiles to return / jump / conditional jump
 - “Fall-through” semantics of assembly blocks is never used

Generating code from a CFG

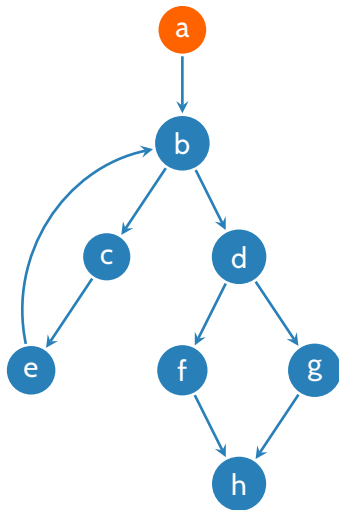
- Simple strategy: terminator always compiles to return / jump / conditional jump
 - “Fall-through” semantics of assembly blocks is never used
- More efficient strategy: elide jumps by ordering blocks appropriately
 - A *covering set of traces* is a set of traces such that
 - Each trace is a simple path (loop free)
 - Each basic block belongs to a trace
 - Any covering set of traces corresponds to a (partial) ordering of blocks, which may elide *some* jumps.

Generating a covering set of traces

Basic algorithm: depth-first traversal of the CFG

- If at least one successor is *unvisited*, elide jump and place the successor next in sequence
- If all successors are visited, terminate branch

(see `codegen_cfg_trace` in `let3.ml`)

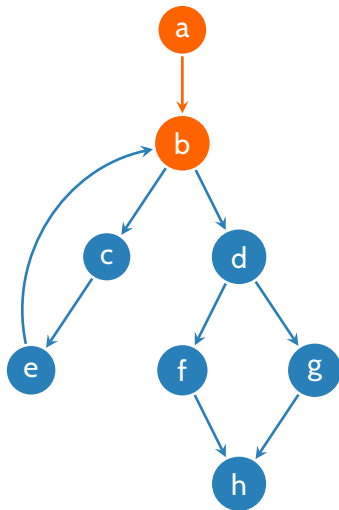


Generating a covering set of traces

Basic algorithm: depth-first traversal of the CFG

- If at least one successor is *unvisited*, elide jump and place the successor next in sequence
- If all successors are visited, terminate branch

(see `codegen_cfg_trace` in `let3.ml`)

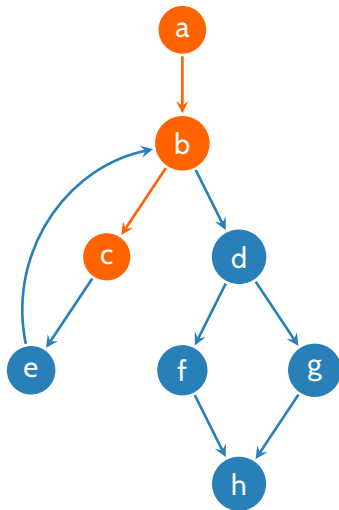


Generating a covering set of traces

Basic algorithm: depth-first traversal of the CFG

- If at least one successor is *unvisited*, elide jump and place the successor next in sequence
- If all successors are visited, terminate branch

(see `codegen_cfg_trace` in `let3.ml`)

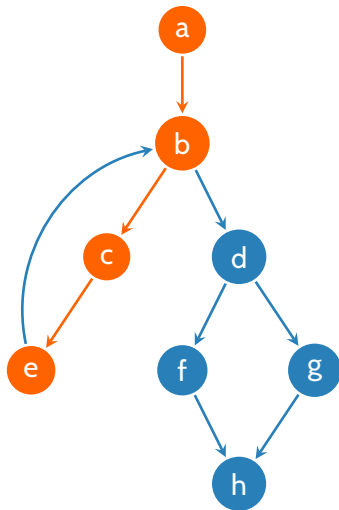


Generating a covering set of traces

Basic algorithm: depth-first traversal of the CFG

- If at least one successor is *unvisited*, elide jump and place the successor next in sequence
- If all successors are visited, terminate branch

(see `codegen_cfg_trace` in `let3.ml`)

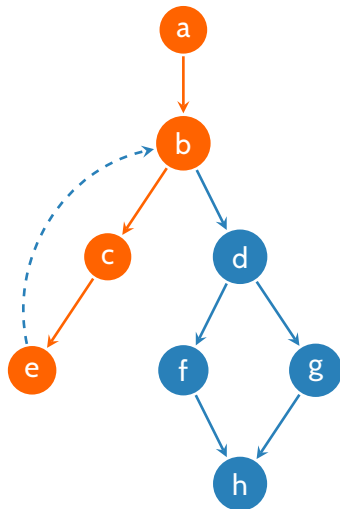


Generating a covering set of traces

Basic algorithm: depth-first traversal of the CFG

- If at least one successor is *unvisited*, elide jump and place the successor next in sequence
- If all successors are visited, terminate branch

(see `codegen_cfg_trace` in `let3.ml`)

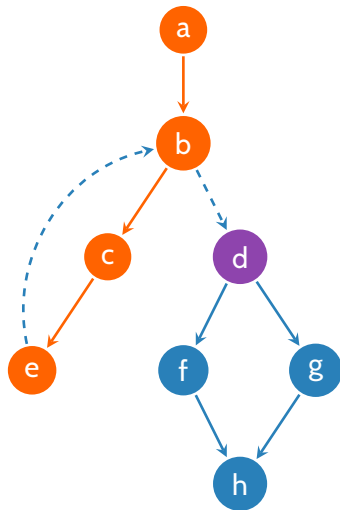


Generating a covering set of traces

Basic algorithm: depth-first traversal of the CFG

- If at least one successor is *unvisited*, elide jump and place the successor next in sequence
- If all successors are visited, terminate branch

(see `codegen_cfg_trace` in `let3.ml`)

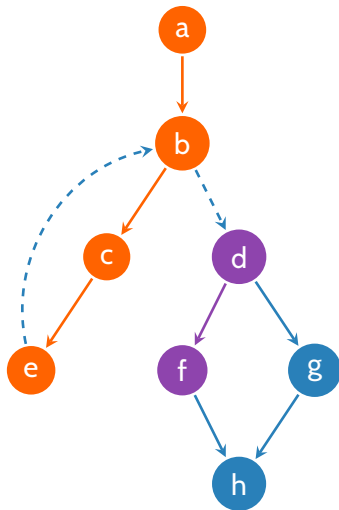


Generating a covering set of traces

Basic algorithm: depth-first traversal of the CFG

- If at least one successor is *unvisited*, elide jump and place the successor next in sequence
- If all successors are visited, terminate branch

(see `codegen_cfg_trace` in `let3.ml`)

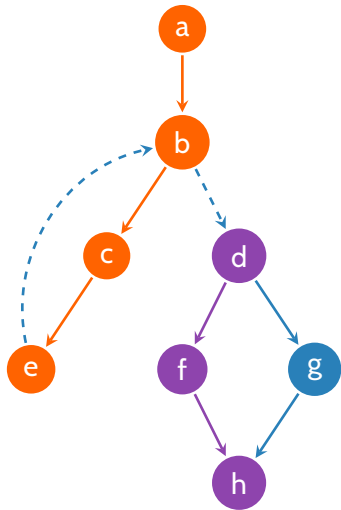


Generating a covering set of traces

Basic algorithm: depth-first traversal of the CFG

- If at least one successor is *unvisited*, elide jump and place the successor next in sequence
- If all successors are visited, terminate branch

(see `codegen_cfg_trace` in `let3.ml`)



Generating a covering set of traces

Basic algorithm: depth-first traversal of the CFG

- If at least one successor is *unvisited*, elide jump and place the successor next in sequence
- If all successors are visited, terminate branch

(see `codegen_cfg_trace` in `let3.ml`)

