252-0210-00L Compiler Design

Sept. 18th - Lecture 1: Introduction: Compilers, Interpreters, OCaml

- Why study compilers?
    - learn (a lot)
        - practical applications of theory
        - lexing / parsing / interpreters
        - how high-level languages are implemented
        - Intel x86 architecture
        - GCC & LLVM
        - Deeper understanding of code, programming language semantics & types
        - Functional programming in OCaml
        - Manipulate complex data structures
        - be a better programmer
- Workload
    - very challenging, implementation-oriented course
        - programming projects can take *tens* of hours per week
- The compiler project
    - Course projects
        - HW1: OCaml programming
        - HW2: X86lite interpreter
        - HW3: LLVMlite compiler
        - HW4: Lexing, parsing, simple compilation
        - HW5: Higher-level features
        - HW6: Analysis and optimizations
    - Goal: Build a *complete compiler* from a high-level, type-safe language to x86 assembly
- Why OCaml

        - dialect of ML - "Meta Language"
            - designed to enable easy manipulation of *abstract syntax trees*
            - Type-safe, mostly pure, functional language with support for polymorphic (generic) algebraic datatypes, modules, and mutable state
            - The OCaml compiler itself is well engineered (can study its source)
        - Haven't learned OCaml?
            - Next couple lectures (& the first exercise session) will introduce it
            - First 2 projects will help you get up to speed programming
            - "Introduction to Objective Caml" by Jason Hickey
- HW1: Hellocaml
        - Available on the course Moodle site
            - individual project - no groups

Lecture

- What is a compiler?
        - e.g. GCC, https://godbolt.org/
        - A program translating one programming language to another (usually high-level to low-level)
            - Typically: high-level source code to low-level machine code (object code)
            - Not always: source-to-source translators, Java bytecode compiler (javac), GWT (Google Web Kit) Java to JavaScript
- Source code
        - optimized for human readability
            - expressive: matches human ideas of grammar / syntax / meaning
            - redundant: more information than needed to help catch errors
            - abstract: exact computation possibly not fully determined by code
- Low-level code
        - Optimized for hardware
            - machine code hard for people to read

        - redundancy, ambiguity reduced
        - abstractions & information about intent is lost
    - Assembly language
        - then machine language
- How to translate?
    - Source code & machine code mismatch
    - Some languages are farther from machine code than others
    - Goals of translation
        - source level expressiveness for the task
        - best performance for the concrete computation
        - reasonable translation efficiency ($< O(n^3)$)
        - maintainable code
        - correctness
- Correct compilation
    - programming languages describe computation precisely
        - translation can be precisely described
        - a compiler can be correct with respect to source and target language semantics
    - Importance
        - broken compilers generate broken code
        - hard to debug source programs if the compiler is incorrect
        - failure has dire consequences for development cost, security, etc
    - Some techniques for building correct compilers (tests)
    - e.g. LLVM Bug #14972 (miscompilation, wrong code), GCC Bug #101797 (ICE: internal compiler error), Compiler hang (slow compilation), Missed optimizations,
- Idea: translate in steps
    - Compile via a series of program representations
    - Source code (character stream) $\rightarrow$ lexical analysis $\rightarrow_{\text{token stream}}$ parsing (front end, machine independent) $\rightarrow_{\text{abstract syntax tree}}$ intermediate code generation (middle end, compiler dependent) $\rightarrow_{\text{intermediate code}}$ code generation (back end, machine dependent) $\rightarrow$ assembly code
    - Intermediate representations are optimized for program manipulation of various kinds
        - semantic analysis: type checking, error checking

        - optimization: dead-code elimination, common subexpression elimination, function inlining, register allocation
        - code generation: instruction selection
    - Representations are more machine specific, less language specific as translation proceeds
    - Lexing, parsing, disambiguation, semantic analysis, translation, control-flow analysis, data-flow analysis etc
- OCaml
    - Distinguishing characteristics
        - Functional & (mostly) "pure"
            - programs manipulate values rather than issue commands
            - functions are first-class entities
            - results of computation can be "named" using `let`
            - has relatively few "side effects" (imperative updates to memory)
        - Strongly and statically typed
    - Most important features
        - Types, Concepts (pattern matching, recursive functions over algebraic datatypes), Libraries
- Interpreters
    - Factorial: everyone's favorite function
        - Consider this implementation of factorial in a hypothetical programming language

```
X = 6;
ANS = 1;
whileNZ(x) {
    ANS = ANS * X;
    X = X + -1;
}
```

        - Need to represent the **abstract syntax** (hide the irrelevant of the concrete syntax)
        - Implement the **operational semantics** (define the behavior, or meaning, of the program)
- OCaml Demo (simple.ml)

Sept. 20th - Lecture 2: OCaml Crash Course: Translating Simple to OCaml

- Interpreters (How to represent programs as data structures. How to write programs that process programs.)
- Optimizing interpreter ($P_0 \to P_1 \to \cdots \to P_n$, and feed P_n to the interpreter)
    - Removing redundancy: `x = 0; x = 0; x = 0; ...`
    - Calculate before giving to interpreter: `x = 2 + 3;` $\to$ `x = 5;`
    - Reduce branches: `ifNZ 0 then c1 else c2` $\to$ `c2`
    - Principle: Divide and Conquer
    - Inlining?
    - phase-ordering problem (choosing the order of optimization)
- $P \to$ `interpreter` $\to$ `result*`
- $P \to$ `translator` $\to$ `OCaml code` $\to$ `ocaml` $\to$ `result*`

Sept. 23rd - Exercise 1: OCaml tutorial

- Exercise: happen biweekly, check dates on Moodle, recorded
- TA will
    - go over important notes introduced in lectures
    - explains solutions of exercise questions (exercise questions introduced one week prior, not graded)
    - introduce the homework assignments
    - answer questions
- Homework Assignment
    - One task biweekly, check dates
    - Friday 13:59 p.m.
    - Late with 30 min: only half of the score
    - Cannot submit beyond 30 min after the due
    - Expected to
        - attend the exercise session to better understand the assignment
        - do the assignment alone (only 1st assignment) or with your teammate
        - check your solution using the **provided docker image**
        - submit solution on Moodle

        - Grading
            - all the gradings are **done automatically in the docker**
            - provided test + hidden tests (100 in total)
            - solution that cannot be compiled in the docker get zero points
        - Notes
            - will detect cheating solutions using our automated scripts
            - plagiarism lead to zero points of the whole **course**
- Exam
    - 120 min written exam
    - past questions/solutions will be available
- Forum
    - Moodleoverflow
        - post questions encountered in the assignment or the course
        - TA will try to answer the questions on weekdays
        - encouraged to answer questions
    - Team forming

OCaml Crash Course

- OCaml
    - Functional language
    - Algebraic data types
    - Pattern matching, etc
- Strict Typing
    - same type operation
    - explicit type casting
- Variables
    - Keyword `let` binds identifiers to expressions
    - Nestings

```
let ans =
    let b = 7 in
        let a = 5 + b in
            a + b
```

- Shadowing

```
let x = 1 in
    let x = 2 in
        x
```

- e.g.

```
let a = 2;;
let b = 3;;

let ans =
    let c = a + b in
        let a = 10 in (* shadows a = 2 *)
            let b = c in (* shadows b = 3 *)
                let b = a in (* shadows b = c*)
                    a + b + c;; (* a = 10, b = 10, c
= 5*)
```

- Functions
  - Functions with a single argument

```
let square (i: int): int = i * i;;
```

- Tuples
  -
```
let tup = 100, "Compilers are fun", true;;
```

  -
```
let second (_, x, _) = x
```

- Custom types and polymorphism
  - `'a`: some type
- Lists
  -
```
let l = [1; 2; 3]
let l' = 1::2::3::[]
```

- Pattern Matching
  -
```
match e with
    |   p1 -> e1
    |   p2 -> e2
```

- Custom Types
    - Types can be recursive and generic
- Task 1: Ackermann function
    - Implement the Ackermann function in OCaml
    - function signature: `ackermann (m: int) (n: int): int`
    - $$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,1) & m > 0, \text{and } n = 0 \\ A(m-1, A(m,n-1)) & m > 0, n > 0 \end{cases}$$
    - ```
      let rec ackermann (m: int) (n: int): int =
          if m = 0 then n + 1
          else if m > 0 && n = 0 then ackermann (m - 1) 1
          else if m > 0 && n > 0 then ackermann (m - 1)
      (ackermann m (n - 1))
          else failwith "invalid"
      ```
        - use `rec` to tell OCaml that the function is recursive
- Task 2: Arithmetic Expression Tree
    - 2.1: Define types `arithm_ast` and `visit_order` to represent the AST and the traversal options
        - ```
          type arithm_ast =
              |   Node of int
              |   Add of arithm_ast * arithm_ast
              |   Mul of arithm_ast * arithm_ast

          type visit_order =
              |   Infix
              |   Prefix
              |   Postfix
          ```
    - 2.2: Implement a function that returns a string representation according to the selected traversal
        - ```
          let as_string (exp: arithm_ast) (order:
          visit_order): string =
              match exp with
                  |   Node x -> string_of_int x
                  |   Add (x, y) ->
                          begin match order with
          ```
          ```

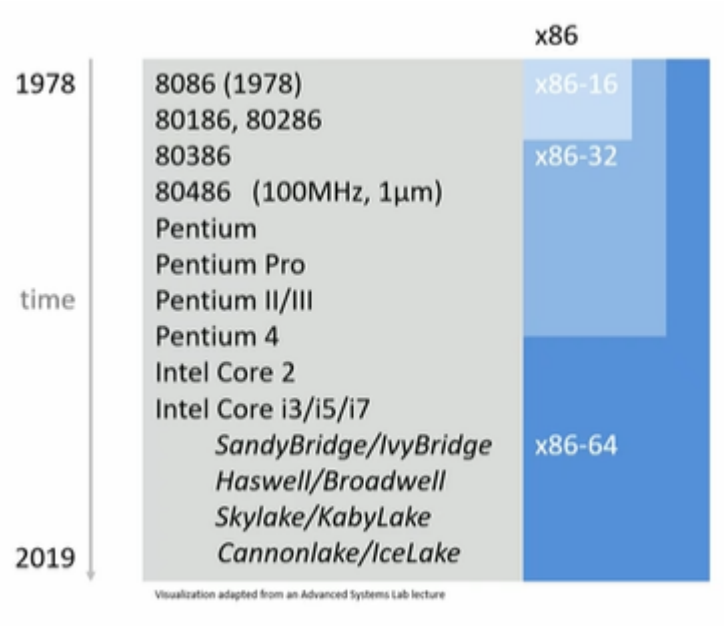
```
                    |   Infix -> "(" ^
(as_string x order) ^ "+" ^ (as_string y
order) ^ ")"
                    |   Prefix ->
                    |   Postfix ->
        |   Mul (x, y) ->
                begin match order with
                    |   Infix -> "(" ^
(as_string x order) ^ "+" ^ (as_string y
order) ^ ")"
                    |   Prefix ->
                    |   Postfix ->
let e = Add (Node 5, Mul (Node 3, Node 7))
```

    - 2.3: Implement a function that evaluates the expression
- Task 3: String Concatenation and Tail Recursion
    - 3.1: Implement a function that concatenates a list of strings, delimiting them with a comma
    - Tail Recursive functions
        - tail recursion is a specific kind of recursion where the value produced by a recursive call is **returned directly by the caller without further compuation**
        -
```
let rec sum l =
    match l with
        |   [] -> 0
        |   h::t -> h + (sum t)

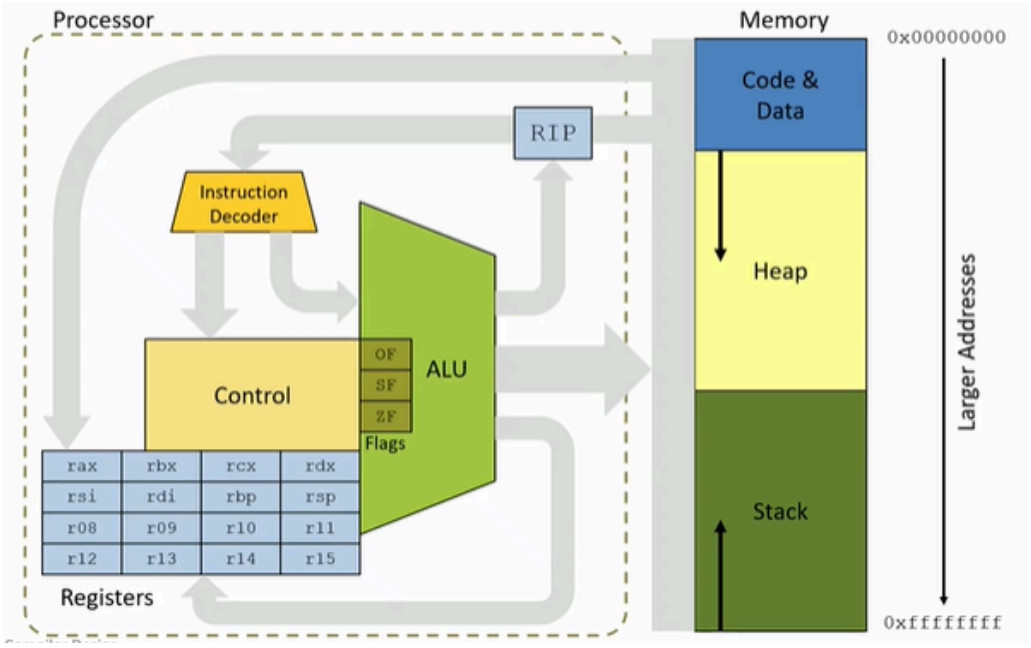
(* tail recursive *)
let rec sum l acc =
    match l with
        |   [] -> acc
        |   h::t -> sum t (h + acc)
```

- Recap
    - Start with HW1
    - OCaml Basics
    - Task 1, 2, 3

Sept. 25th - Lecture 3: x86lite

- Back end
    - Compiler Intermediate Representation → Assembly code / ISAs
- A detour through a compiler
    - Compiler explorer
    - ```
      long foo(long i, long j) {
          return i * j;
      }
      ```
        - ```
          foo:
              movq %rdi, %rax
              imulq %rsi, %rax
              retq
          ```
    - x86
        - Binary compatibility (old code runs on new hardware)
        - Complex ISA (1500+ instructions)
        - Multiple extensions
        - Non-Intel implementations (AMD, Via)
        - 

    - x86lite
        - simple subset of x86
        - only 64-bit signed integers (no floating point, no 16-bit, etc)
        - only about 20 instructions

        - sufficient as a target language for general-purpose computing
- x86lite
    - 

    - Registers
        - hold values: integers, addresses
        - `rbp, rsp`: base pointer & stack pointer, used for call-stack manipulation (for function calls)
        - `RIP`: instruction pointer, holds the address of the next instruction
    - Processor state
        - `OF, SF, ZF`: overflow flag, sign flag, zero flag
        - overflow: set when the result is too big/small to fit in 64-bit reg
        - sign: set to the sign of the result (0 = positive, 1 = negative)
        - zero: set when the result is 0
    - Code & data
        - The actual program instructions, $\texttt{INSTR}_1, \ldots$
        - program constants & globals
    - Stack
        - Used for function calls and local variables|
    - Heap
        - Dynamically allocated memory (via calls to `malloc`)
    - Arithmetic & control flow
    - Memory model

        - it consists of 2^{64} bytes numbered `0x00000000` through `0xffffffff`
        - x86lite treats the memory as consisting of 64-bit (8-byte) quadwords (all memory addresses are evenly divisible by 8)
        - By convention, the stack grows from high addresses to low addresses
    - The register `rsp` points to the top of the stack
        - `pushq SRC`: `rsp` ← `rsp - 8`; `Mem[rsp]` ← `SRC`
        - `popq DST`: `rsp` ← `rsp + 8`; `DST` ← `Mem[rsp]`
- C vs. x86lite
    - 64-bit integers (`long`) vs. 64-bit registers
    - unlimited variables vs. fixed number of registers
    - complex statements vs. simple instructions
- x86lite cont.d
    - `mov`
        - `movq SRC, DST`
        - move (copy) the source into destination
        - DST is treated as a **location**
            - a location can be a register or a memory
        - SRC is treated as a value
            - contents of a register or memory address
            - can also be an immediate value
    - AT&T notation: source **before** destination
        - prevalent in the Unix ecosystems
        - Immediate values prefixed with $
        - Registers prefixed with %
        - Mnemonic suffixes
            - `q`: quadword (4 words)
            - `l`: long (2 words)
            - `w`: word (16-bit)
            - `b`: byte (8-bit)
    - Intel notation: destination **before** source
        - used in the Intel specification / manuals
        - prevalent in the Windows ecosystem

        - Instruction variant determined by register name
- Operands
        - Registers: one of the 16 registers, the value of a register is its contents
        - Immediate: 64-bit literal signed integer
        - Label: a "label" representing a machine address, the assembler/linker/loader resolves labels
        - Memory address
- Arithmetic instructions
        - negation, addition, subtraction, multiplication
        - `negq DST; addq SRC, DST; subq SRC, DST; imulq SRC, REG`
- Logic/bit manipulation instructions
- Code blocks & Labels
        - x86 assembly code is organized into labeled blocks
        - Labels indicate code locations that can be jump targets (either through conditional branch instructions or function calls)
        - Labels are translated away by the linker and loader - instructions live in the "code segment"
        - An x86 program begins executing at a designated code label (usually "main")
- Jumps, call and return
        -
```
void bar() {
    //...
}

void foo() {
    //...
    bar();
}
```

-
```
bar:
    ...
    ret

foo:
    ...
    call bar
    ...
    ret
```
- 1st argument in `%rdi`, 2nd argument in `%rsi`, 3rd argument in `%rdx`, return value in `%rax`

- Code examples
  -
```
long foo(long * arr, long i, long n) {
    if (i < n) {
        return arr[i];
    } else {
        return -1l;
    }
}
```
  -
```
foo:
    movq $-1, %rax
    cmpq %rdx, %rsi
    jge .LBB0_2
    movq (%rdi, %rsi, 8), %rax
.LBB0_2:
    retq
```

- x86lite addressing
  -
```
long a[0, 42, 2020];

long b = (long)a;
long b = *a;
long b = *(a + 2);

long c = 1;
long b = a[c];
long b = a[c + 1];
```

```
// Array [0, 42, 2020]
// Array address 0xBEEF

movq %0xBEEF, %rax

movq %rax, %rbx     // rbx = 0xBEEF
movq (%rax), %rbx   // rbx = 0
movq 16(%rax), %rbx // rbx = 2020

movq $1, %rcx
movq (%rax, %rcx), %rbx     // rbx = 42
movq 8(%rax, %rcx), %rbx    // rbx = 2020
```

- In general, we have base, index, and displacement/offset
- `addr(ind) = Base + [Index * 8] + Disp`

Sept. 27th - Lecture 4: x86lite Programming / C calling convention

- Stack frame

```
pushq %rbp      // save the old rbp, need for restore it before returning
movq %rsp, %rbp // update rbp to point to the base of the new frame
subq $16, %rsp  // allocate stack space by moving rsp

(other code)

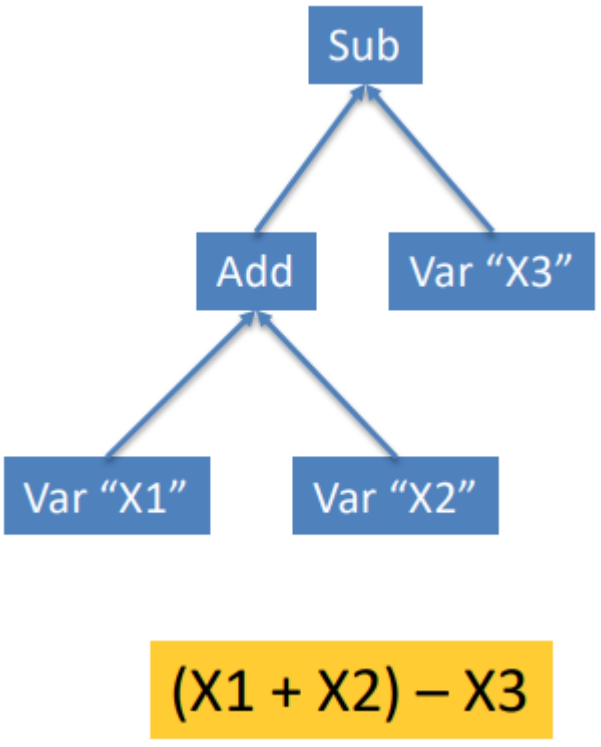
addq $16, %rsp  // deallocate stack space by moving rsp
popq %rbp       // restore old rbp
```

  - `rsp` points to the top of the stack frame, and `rbp` points the bottom of the stack frame
- Manipulating the stack
  - `pushq, popq`
- Local/temporary variable storage
  - need space to store

        - global variables
        - values passed as arguments to procedures
        - local variables
            - either defined in the source program, or
            - introduced by the compiler
    - processors provide two options
        - registers: fast, small size, very limited number
        - memory: slow, very large amount of space (caching is important)
    - In practice in x86
        - registers are limited (and have restrictions)
        - divide memory into regions including stack and the heap
- Calling conventions
    - specify the locations (e.g. register or stack) of arguments
        - passed to a function, and
        - returned by the function
    - designate registers
        - caller save - freely usable by the called code
        - callee save - must be restored by the called code
    - define the protocol for deallocating stack-allocated arguments
        - caller cleans up
        - callee cleans up
- x86-64 system v amd 64 abi

- More modern variant of C calling conventions
  - Used on Linux, Solaris, BSD, OS X
- Callee save: `rbp, rbx, r12-r15`
- Caller save: all others
- Parameters
  - 1..6: `rdi, rsi, rdx, rcx, r8, r9`
  - 7+: on the stack (in right-to-left order)
  - Thus, for n > 6, the nth argument is at `((n-7)+2)*8 + rbp`
- Return value in `rax`
- 128 byte "red zone" – scratch pad for the callee's data
  - For optimization purposes, but how?

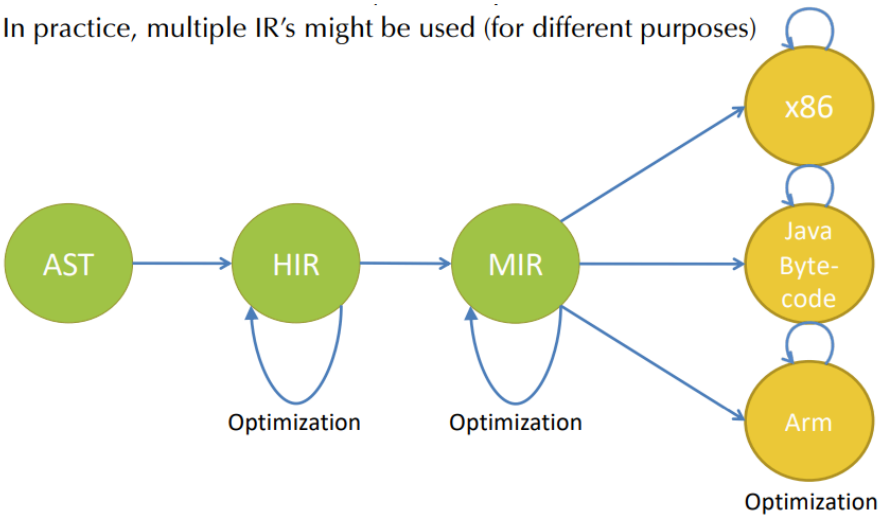
Oct. 2nd - Lecture 5: Intermediate Representations

- "Lowering" an AST to ASM Code
  - 

    - Direct lowering into a segment of asm code (map from source level to assembly level directly)
- Directly generating x86
  - Directly translating AST to assembly
    - For simple languages, no need for intermediate representations
      - e.g. `(X1 + X2) - X3`

        - Maintain invariants
            - code emitted for a given expression computes the answer into `rax`
        - Key challenges
            - storing intermediate values needed to compute complex expressions
            - some instructions use specific register
    - One simple strategy
        - To compile: `e1 op e2`
            - recursively compile its sub-expressions
            - process the results
        - Invariants
            - compilation of an expression yields its result in `rax`
            - argument (`Xi`) is stored in a dedicated operand
            - Intermediate values are pushed onto the stack
            - stack slot is popped after use (so the space is reclaimed)
        - Resulting code is wrapped to comply with cdecl calling conventions
- Intermediate representations
    - Why do smth else?
        - We followed **syntax-directed** translation
            - input syntax **uniquely** determines the output (no complex analysis or code transformation)
            - it works fine for simple languages
        - The resulting code quality is poor
        - Richer source language features are hard to encode (structured data types, objects, first-class functions)
        - It is hard to optimize the resulting assembly code
            - the representation is too concrete (has committed to using certain registers and the stack)
            - only a fixed number of registers
            - some instructions restrict where operands located
        - Retargeting the compiler to a new architecture is hard
        - Control-flow is not structured

        - arbitrary jumps
        - implicit fall-through makes sequences of code non-modular
- Abstract machine code
    - hide details of the target architecture
    - allows machine independent code generation and optimization
- Multiple IRs
    - Goal: get program closer to machine code without losing the information needed to do analysis and optimizations
    - In practice, multiple IR's might be used (for different purposes)



- What is a good IR
    - easy translation target (from the level above)
    - easy to translate (to the level below)
    - narrow interface (fewer constructs means simpler phases/optimizations)
- Example: source language have `while, for, foreach` loops
    - IR might have only `while` loops and sequencing
    - Translation eliminates `for, foreach`
    - ```
      [for (pre; cond; post) {body}] == [pre; while (cond) {body}]
      ```

- IRs at the extreme
    - High-level IR
        - abstract syntax + new node types not generated by the parser (type checking information or disambiguated syntax nodes)
        - typically preserves the high-level language constructs
      ```

        - allows high-level optimizations based on program structure
        - useful for semantic analysis like type checking
    - Low-level IR
        - machine dependent assembly code + extra pseudo-instructions
        - source structure of the program is lost
        - allows low-level optimizations based on target architecture
    - Mid-level IR
        - Intermediate between AST and assembly
        - may have unstructured jumps, abstract registers or memory locations
        - convenient for translation to high-quality machine code
        - ```
          (X1 + (X2 - X3)) * X4

          t1 = X2 - X3
          t2 = X1 + t1
          t3 = X4 * t2
          ```
        - examples
            - triples: `OP a b`
                - useful for instruction selection on x86 via "tiling"
            - quadruples: `a = b OP c` (three address form)
            - SSA (static single assignment): variant of quadruples where each variable is assigned exactly once
                - easy dataflow analysis for optimization
                - e.g. LLVM: industrial-strength IR, based on SSA
            - stack-based
                - easy to generate (e.g. Java Bytecode)
- Growing an IR
          ```

- Start: a very simple IR for the arithmetic language
    - very high level
    - no control flow
- Goal: a simple subset of the LLVM IR
    - LLVM = "low-level virtual machine"
- Add features needed to compile rich source language

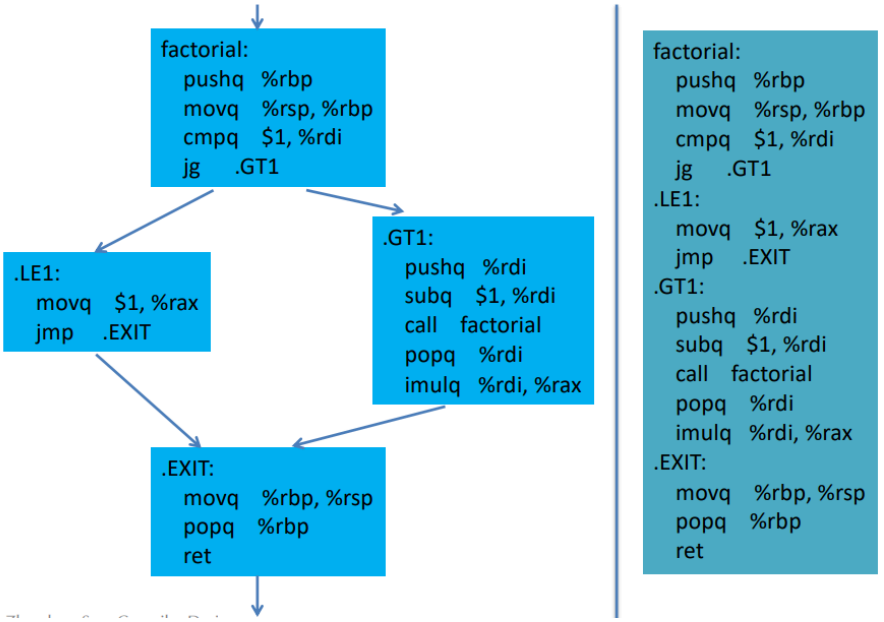
- Simple LET-based IR
    - Eliminating nested expressions
        - e.g. `((1 + X4) + (3 + (X1 * 5)))`
        - fundamental problem: compiling complex & nested expression forms to simple operations
        - idea: name intermediate values, make order of evaluation explicit
        - • Given

```
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
```

    - IR1: Expressions
    - IR2: Commands
    - IR3: Local control flow
    - IR4: Procedures (top-level functions)
    - Basic blocks and CFGs (control flow graphs)
        - **basic block**: a sequence of instructions that is always executed starting at the first instruction and always exits at the last instruction
            - often starts with a label that names the **entry point** of the basic block
            - often ends with a control-flow instruction: the "link" (e.g. branch/return), or encountering a label
            - 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 block A **might** jump to the label of block B
- 

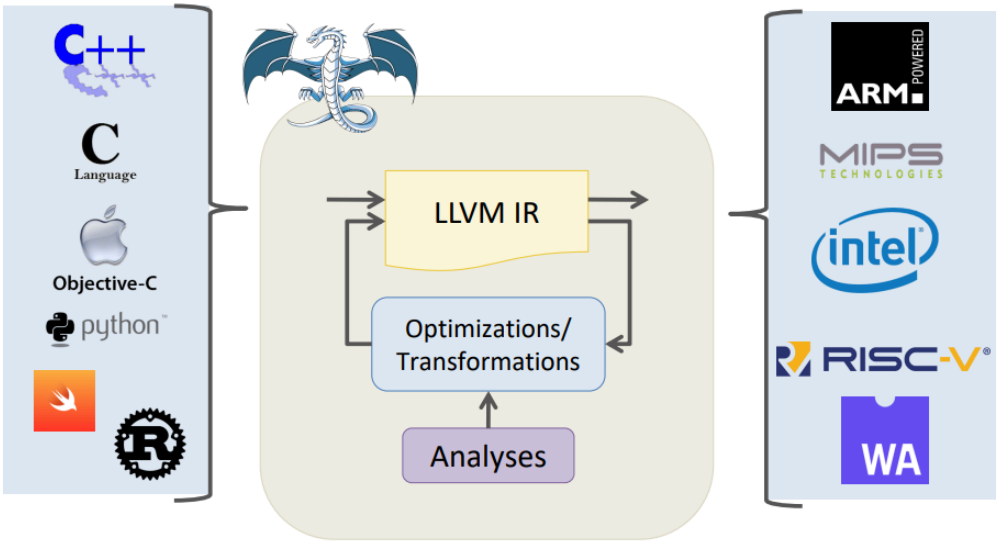
Oct. 4th - Lecture 6: Intermediate Representation II

Oct. 7th - Exercise Session 2: x86lite

- Assignment instruction
    - Due: 18 Oct, 13:59
    - max two people, can work alone
    - automatic grading with prepared testcases, ensure the project can be correctly compiled
    - Goal: assembler and simulator for executing x86lite
        - Input code → part 2: assembler and loader → part 1: simulator
        - simulator: simulate the execution for a given machine code
            - memory: will use only 64K bytes of memory
            - symbolic instruction encoding: a fixed-length, 8-byte encoding of x86lite
            - operand restrictions: simulator may implement a superset of the x86lite specification by executing instructions with invalid operands

        - termination and system calls: will not simulate system calls. the provided run function will call the step function until %rip contains exit_addr
    - assembler: translate the assembly code into machine code
        - memory: calculate the address for text and data should be loaded according to the memory layout
        - symbol table: record the absolute address of each label definition
        - translator: replace each instruction and data element using symbol table
    - loader: setup the initial execution states
        - memory: create initial memory layout for text and data should be loaded
        - register: initialize all machine registers
        - stack: initialize stack

Oct. 9th - Lecture 7: LLVM (lite) - Low Level Virtural Machine

- The state of C/C++ compilers in the early 2000s: no common infrastructure
- LLVM Compiler Infrastructure
    - 

- LLVMlite factorial

- ```
  long factorial(long n) {
      long acc = 1;
      while (n > 0) {
          acc = acc * n;
          n -= 1;
      }
      return acc;
  }
  ```

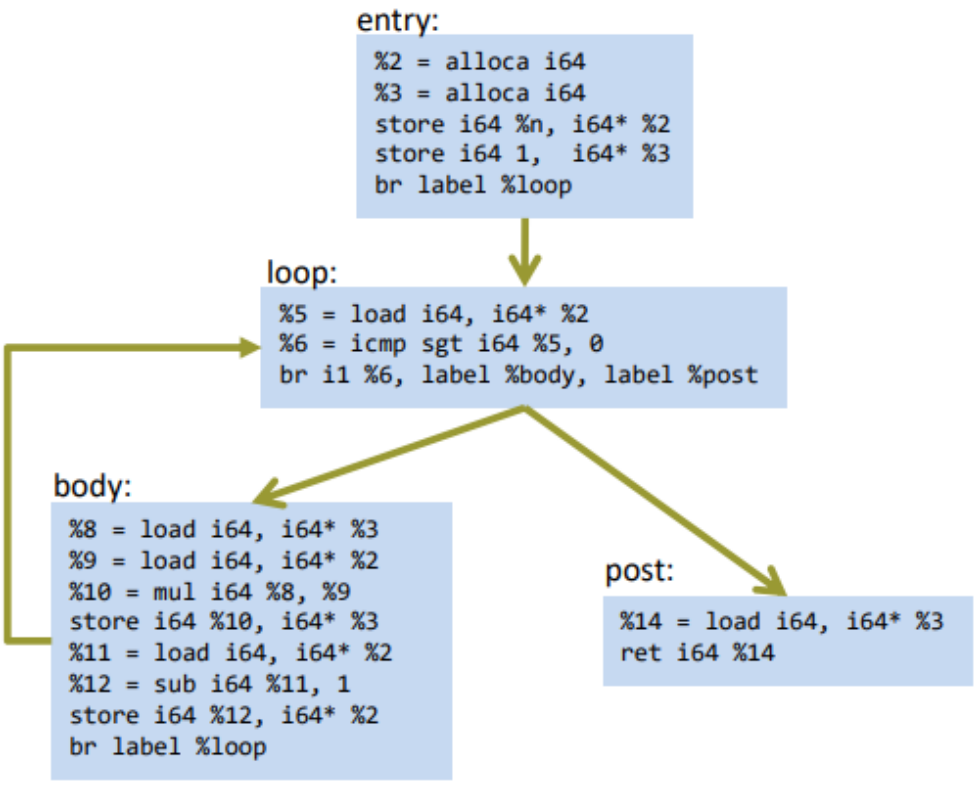
- ```
  define i64 @factorial(i64 %0) {
      // all the code below
  }
  entry:
      %2 = alloca i64              // SSA: single
  static assignment values
      %3 = alloca i64
      store i64 %n, i64* %2
      store i64 1,  i64* %3
      br label %loop

  loop:
      %5 = load i64, i64* %2
      %6 = icmp sgt i64 %5, 0      // sign greater-than
  comparison
      br i1 %6, label %body, label %post

  body:
      %8 = load i64, i64* %3
      %9 = load i64, i64* %2
      %10 = mul i64 %8, %9
      store i64 %10, i64* %3
      %11 = load i64, i64* %2
      %12 = sub i64 %11, 1
      store i64 %12, i64* %2
      br label %loop

  post:
      %14 = load i64, i64* %3
      ret i64 %14
  ```

- Control-flow graphs

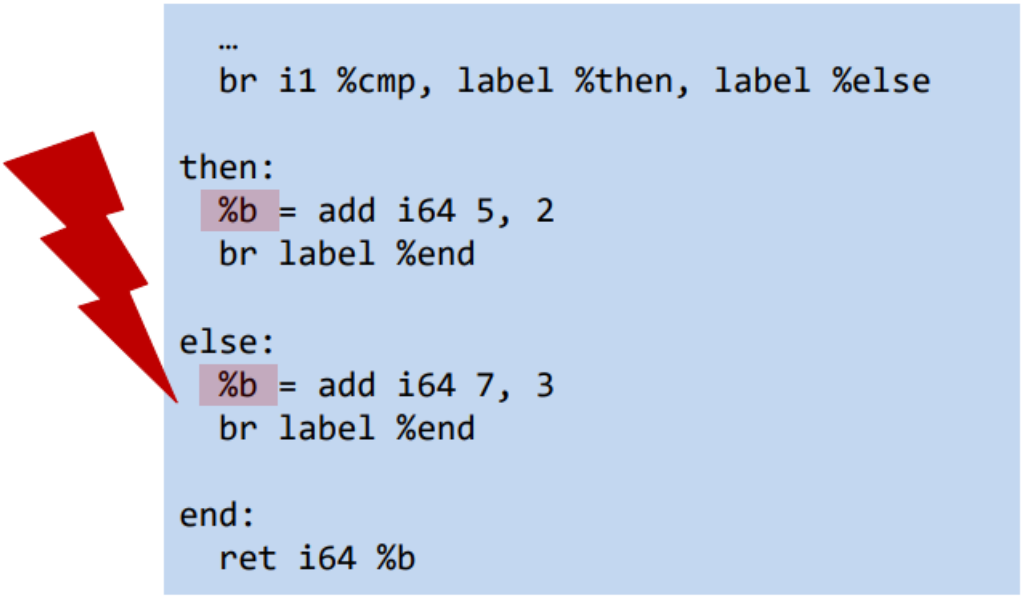


- - `type cfg = block * (lbl * block) list`
- LLVM invariants
    - No two block have the same label
    - All targeted labels are defined among the set of basic blocks
    - There is a distinguished, potentially unlabeled, entry block
- Basic Blocks
    - A sequence of instructions that always executed starting at the first instruction and always exits at the last instruction
        - start with a label that names the **entry point** of the basic block
        - ends with a control-flow instruction (branch or return), a so-called **terminator**
        - contains no other control-flow instructions
        - contains no interior label used as a jump target
    - OCaml representation

```
type block  = {
    insns   : (uid * insn) list;
    term    : (uid * terminator)
}
```

- Storage Model
    - Local variables

        - defined by instructions (e.g. `%10 = mul i64 %8, %9`)
        - must satisfy the **single static assignment** invariant
            - each %uid appears on the left-hand side of an assignment only once in the entire control flow graph
        - intended to be an abstract version of machine registers
    - Global declarations
        - `@str = [12 * i8] c"Hello World\00"`
    - Heap-allocated structures created by external calls
        - `%ptr = call i8* @malloc(i64 42)`
    - Abstract locations: references to (stack-allocated) storage created by the `alloca` instruction
        - `%ptr = alloca i64`
    - Limitations of SSA



        - Use a memory location for a variable `b`

Structured Data in LLVM

- Array indexing
    -
```
void foo(long* ints) {
    ints[3] = 42;
}

void foo(long* ints, long n) {
    ints[n] = 42;
}
```

```
define void @foo(i64* %0) {
    %3 = getelementptr, i64* %0, i64 3
    store i64 42, i64* %3
    ret void
}

define void @foo(i64* %0, i64 %1) {
    %3 = getelementptr, i64* %0, i64 %1
    store i64 42, i64* %3
    ret void
}
```

- Point struct
    - ```
      struct Point {
          long x;
          long y;
      };

      void foo() {
          struct Point p;
          p.x = 1;
          p.y = 2;
      }
      ```

```
%Point = type { i64, i64 }

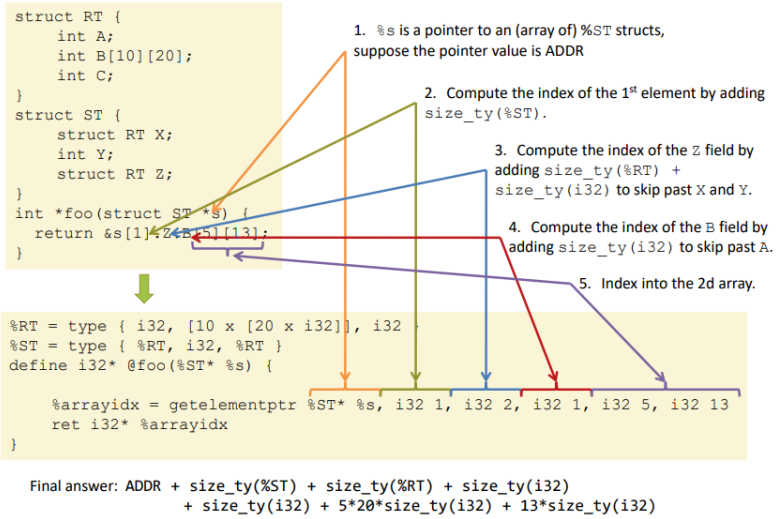
define void @foo() {
    %1 = alloca %Point
    %2 = getelementptr, %Point* %1, i64 0, i64 0
    store i64 1, i64* %2
    %3 = getelementptr, %Point* %1, i64 0, i64* 1
    store i64 2, i64* %3
    ret void
}
```

- Arrays of struct
```

- ```
  struct Point {
      long x;
      long y;
  };

  void foo(struct Point* ps, long n) {
      ps[n].y = 42;
  }
  ```

- `getelementptr`
  - LLVM provides the **`getelementptr`** instruction to compute pointer values
    - given a pointer and a "path" through the structured data pointed to by that pointer, it computes an address
    - this is the abstract analog of the x86 LEA (load effective address). it does not access memory
    - it is a "type indexed" operation, since the size computations depend on the type
    - `<result> = getelementptr <type>* <ptrval>{,<ty> <idx>}*`
  - GEP **never** dereferences the address it's calculating
    - GEP only produces pointers by doing arithmetic
    - It doesn't really traverse the links of a data structure
  - To index into a deeply nested structure, need to "follow the pointer" by loading from the computed poitner
  - **GEP Example***



- Struct parameters & return values
  - // smth
  ```

Oct. 11th - Lecture 8: LLVMlite Cont.d

- Compiling LLVMlite to x86
    - LLVMlite arithmetic instructions (`add, sub, mul`), bit instructions (`and, or, xor, shl, lshr, ashr`), control flow instructions (`call, ret, br`), memory instructions (`load, store, alloca`), and other miscellaneous instructions (`icmp, getelementptr, bitcast`)
    - Quadwords, raw `i8` values are not allowed (only manipulated via pointer), array and struct types are laid out sequentially in memory
    - Compiling LLVM locals
        - How do we manage storage for each `%uid` defined by an LLVM instruction?
        - Option 1
            - Map each uid to a register
            - efficient, but difficult to do effectively, many uid values, only 16 registers
        - Option 2
            - Map each uid to a stack-allocated space
            - less efficient, but simple to implement
    - C → LLVMlite → x86
        -
```
long foo(long a, long b) {
    return a + b;
}
```

```
define i64 @foo(i64 %a, i64 %b) {
    %0 = add i64 %a, %b
    ret i64 %0
}
```

```
foo:
    pushq %rbp
    movq %rsp, %rbp
    subq $24, %rsp

    movq %rdi, -24(%rbp)
    movq %rsi, -16(%rbp)
    movq -16(%rbp), %rax
    movq -24(%rbp), %rcx
```

```
    addq %rcx, %rax
    movq %rax, -8(%rbp)

    movq -8(%rbp), %rax

    addq $24, %rsp
    popq %rbp
    retq
```

- Compared to:

```
add_asm:
    movq %rdi, %rax
    addq %rsi, %rax
    retq
```

- We are generating x86 programs in a principled, general way

- `getelementptr` → x86

-
```
struct Point {
    long x;
    long y;
}

void foo() {
    struct Point p;
    p.x = 1;
    p.y = 2;
}
```

```
%Point = type{ i64, i64 }

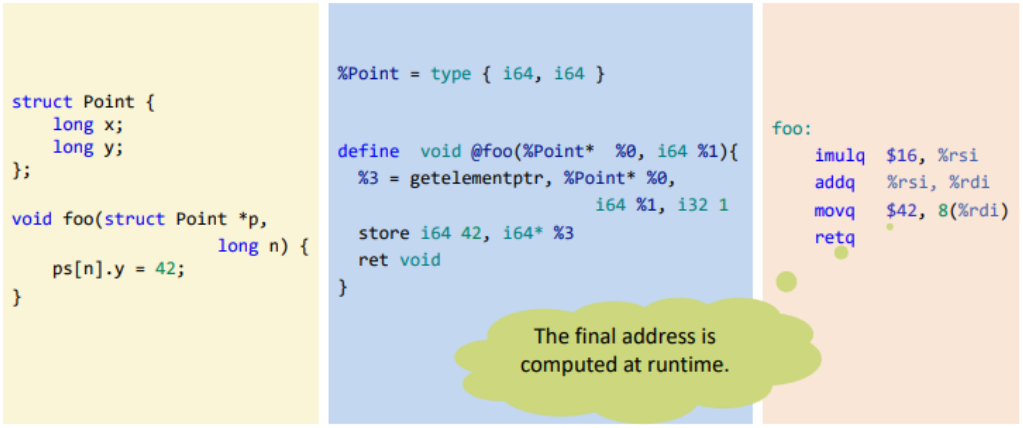
define void @foo() {
    %1 = alloca %Point
    %2 = getelementptr, %Point* %1, i32 0,
i32 0
    store i64 1, i64* %2
    %3 = getelementptr, %Point* %1, i32 0,
i32 1
    store i64 2, i64* %3
    ret void
}
```

```
foo:
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp

    movq $1, -16(%rbp)
    movq $2, -8(%rbp)

    addq $16, %rsp
    popq %rbp
    retq
```

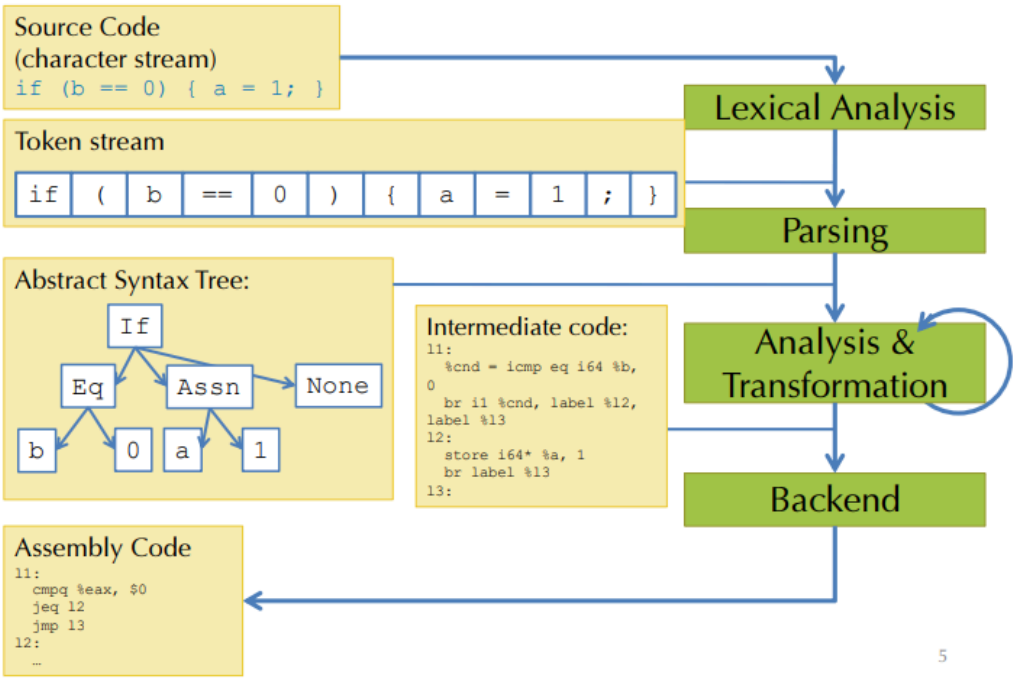
    - %1 in this case corresponds to `-16(%rbp)`, and getelementptr → base + offset
- Compilation of GEP
    - Translate GEP's base pointer into an actual address
    - Compute the offset specified by the indices and add it to the base address
- Array indexing



- If-statements and loops
    - they correspond to branching in the Control Flow Graph
    - Basic Blocks are (mostly) codegen'd independently
    - The resulting BBs are connected via jumps
- Simple x86 code generation for functions
    - Write function prologue
        - setup the stack frame
        - allocate enough space for all alloca's
        - handle arguments
    - For each basic block
        - generate x86 code

        - keep track of the jump targets (other basic blocks)
    - Fill in the jump targets of each basic block
    - Write function epilogue
        - handle return value
        - tear down stack frame

Oct. 16th - Lecture 9: Lexing - DFAs and OCamllex

- Compilation in a Nutshell
    - 

- Lexing (lexical analysis, tokens, regular expressions, automata)
    - First step: lexical analysis
        - Change the **character stream** into **tokens**
        - ```
          if (b == 0) {a = 1;}

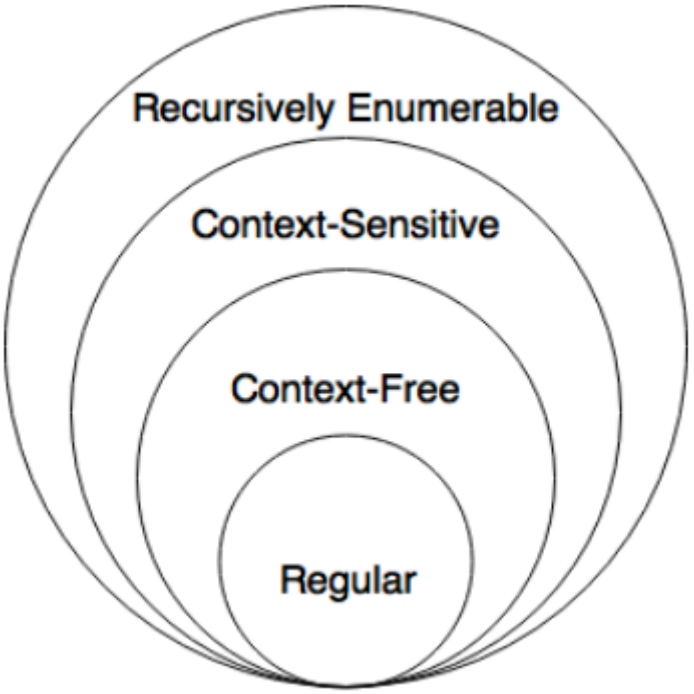
          IF; LPAREN; Ident("b"); EQEQ; Int(0); RPAREN;
          LBRACE; Ident("a"); EQ; Int(1); SEMI; RBRACE
          ```
        - Token: data type that represents **indivisible** "chunks" of text
            - Identifiers, keywords, integers, floating point, symbols, strings, comments
        - Often delimited by **whitespace** (' ', \t, etc) - In Python/Haskell whitespace is significant
    - Lexing by hand
          ```

        - Problems
            - precisely define tokens, matching tokens simultaneously, reading too much input (need to look ahead), error handling, hard to compose/interleave tokenizer code, hard to maintain
    - Examples
        - FORTRAN
            - whitespace insignificant: `var1` = `va r1`
            - How about

              looping: `DO 5 I = 1,25`

              assignment: `DO 5 I = 1.25`
            - Need to scan until "." and "," to determine the type of instruction
        - C++
            - Template syntax: `Foo<Bar>`
            - Stream syntax: `cin >> x`
            - Then, nested templates: `Foo<Bar<Bazz>>`
- Principled solution to lexing
    - Regular expressions (Regex)
        - precisely describe sets of strings
        - Terms
            - ε: stands for the empty string
            - `'a'`: an ordinary character stands for itself
            - $R_1|R_2$: alternatives, stands for choice of R_1 or R_2
            - R_1R_2: concatenation, stands for R_1 followed by R_2
            - R^*: Kleene star, stands for *zero* or more repetitions of R
        - Extensions
            - `"foo"`: strings
            - $R+$: one or more repetitions of R, equivalent to RR^*
            - $R?$: 0 or 1 occurrence of R
            - `['a' - 'z']`: one of lowercases
            - `[^'0'-'9']`: any character except 0 through 9
            - R `as` x: Name the string matched by R as x

- Chomsky Hierarchy



- How to match?
    - Consider input string: `ifx = 0`
        - Either `if, x, =, 0`, or `ifx, =, 0`
    - regex alone can be ambiguous
    - We need a rule for choosing between the options
        - Most languages choose "longest match"
        - So 2nd option above will be picked
        - but the 1st option is correct for parsing
    - Conflicts: tokens whose regex have a shared prefix
    - How to resolve?
        - ties broken by giving some matches higher priority
        - Usually specified by order the rules appear in the lex input file
- Lexer Generator
    - Reads a list of regexs: $R_1, \ldots, R_n$, one per token
    - Each token has an attached "action" A_i (just a piece of code to run when the regular expression is matched)

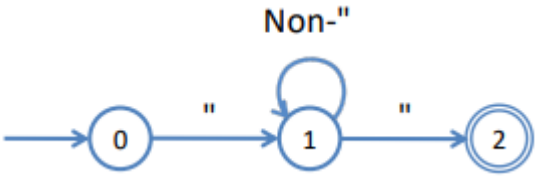
- ```
  rule token = parse
  | '-'?digit+            { Int (Int32.of_string (lexeme lexbuf)) }
  | '+'                   { PLUS }
  | 'if'                  { IF }
  | character (digit|character|'_')*   { Ident (lexeme lexbuf) }
  | whitespace+                        { token lexbuf }
  ```
  token regular expressions

  actions

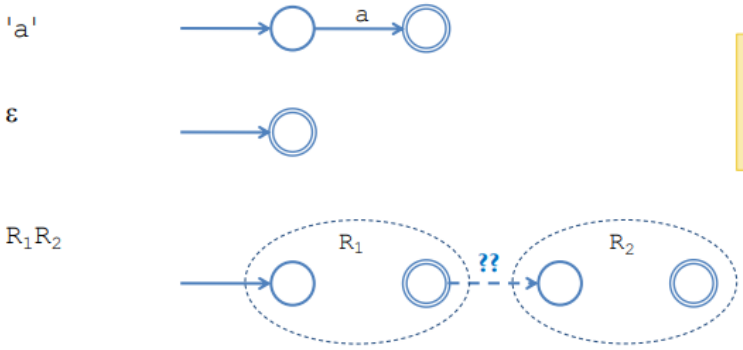
- Generates scanning code that
  - Decides whether the input is of the form $(R_1|\ldots|R_n)^*$
  - After matching a (longest) token, runs the associated action
  - use either "early rule prioritized" or "longest match"
- Implementation Strategies
  - Most tools: lex, ocamllex, flex, etc.
    - Table-based
    - Deterministic Finite Automata (DFA)
    - Goal: Efficient, compact representation, high performance
  - Other approaches
    - Brzozowski derivatives
    - Idea: directly manipulate the (abstract syntax of) the regular expression
    - Compute partial "derivatives"
      - Regex that is "left-over" after seeing the next character
    - Elegant, purely functional, implementation
- Finite Automata
  - Consider regex: `‘"’ [^‘"’]* ‘"’`
  - An automaton (DFA) can be represented as
    - A transition table

	"	Non-"
0	1	ERROR
1	2	1
2	ERROR	ERROR

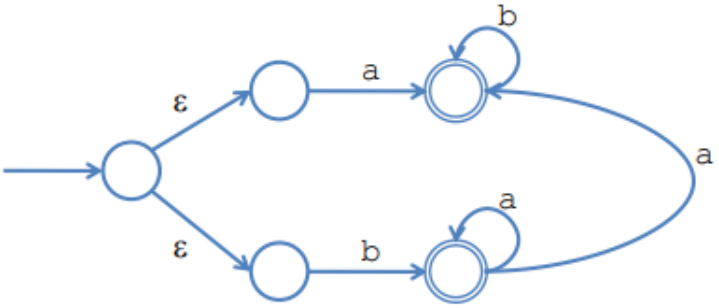
    - A graph
  ```



- Regex to finite automaton
    - For every regex, we can build a finite automaton
    - Strategy: consider every possible regex (by induction on the structure of the regex)

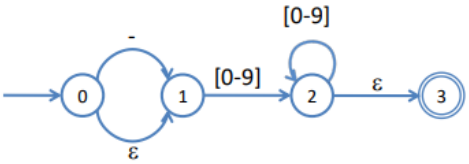


- What about $R_1|R_2$? Nondeterministic Finite Automata
    - A finite set of states, a start state, and accepting state(s)
    - Transition arrows connecting states (labeled by input symbols, or ε)
    - Nondeterministic: two arrows leaving the same state may have the same label

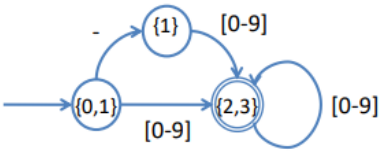


- Regex to NFA
    - Converting regex to NFAs is easy
    - Assume each NFA has one start state, and unique accept state
- DFA vs. NFA
- NFA to DFA conversion (intuition)
    - Idea: Run all possible executions of the NFA "in parallel"
    - Keep track of a set of possible states: "finite fingers"

- Subset construction
- Consider: `-? [0-9]+`
  - NFA representation



  - DFA representation

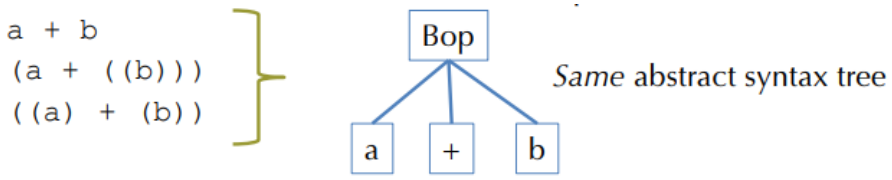


- Summary of Behaviour
  - Take each regex R_i and its action A_i
  - Compute the NFA formed by $(R_1|\ldots|R_n)$
    - remember the actions associated with the accepting states of the `R_i`
  - Compute the DFA for this big NFA
    - there may be multiple accept states
    - a single accept state may correspond to one or more actions
  - Compute the minimal equivalent DFA
    - standard algorithm due to Myhill & Nerode
  - Produce the transition table
  - Implement longest match
    - start from initial state
    - follow transitions, remember last accept state entered
    - accept input until no transition is possible
    - perform the highest-priority action associated with the last accept state; if no accept state there is a lexing error

Oct. 18th - Lecture 10: Parsing

- Syntactic Analysis: Overview
  - Input: stream of tokens
  - Output: abstract syntax tree

- Strategy
    - Parse the token stream to traverse the "concrete" syntax
    - During traversal, build a tree representing the "abstract" syntax
- Why abstract?
    - Consider:



- Note: parsing does not check many things
    - variable scoping, type agreement, initialization

- Specifying Language Syntax
    - How to describe language syntax precisely and conveniently?
    - Tokens use regular expressions, but there are limitations
        - DFA's have only finite number of states
        - DFA cannot count (consider the language of strings with balanced parentheses)
- Context Free Grammars
    - Specification of the language of balanced parentheses

      $S \mapsto (S)S, S \mapsto \varepsilon$
    - The definition is recursive
    - Idea: "derive" a string in the language starting from S and rewriting according to the rules

      $S \mapsto (S)S \mapsto ((S)S)S \mapsto ((\varepsilon)S)S \mapsto ((\varepsilon)S)\varepsilon \mapsto ((\varepsilon)\varepsilon)\varepsilon = (())$
    - We can replace the "nonterminal" S by its definition anywhere
    - A CFG accepts a string iff there is a derivation from the start symbol
    - CFGs mathematically
        - A set of terminals (e.g. a lexical token)
        - A set of nonterminals (e.g. S and other syntactic variables)
        - A designated nonterminal called the **start symbol**
        - A set of **productions**: $LHS \to RHS$
            - LHS is a nonterminal
            - RHS is a string of terminals and nonterminals
    - Another Example

- Consider a grammar that accepts parenthesized sums of numbers

  $S \mapsto E + S \mid E$

  $E \mapsto \text{number} \mid (S)$

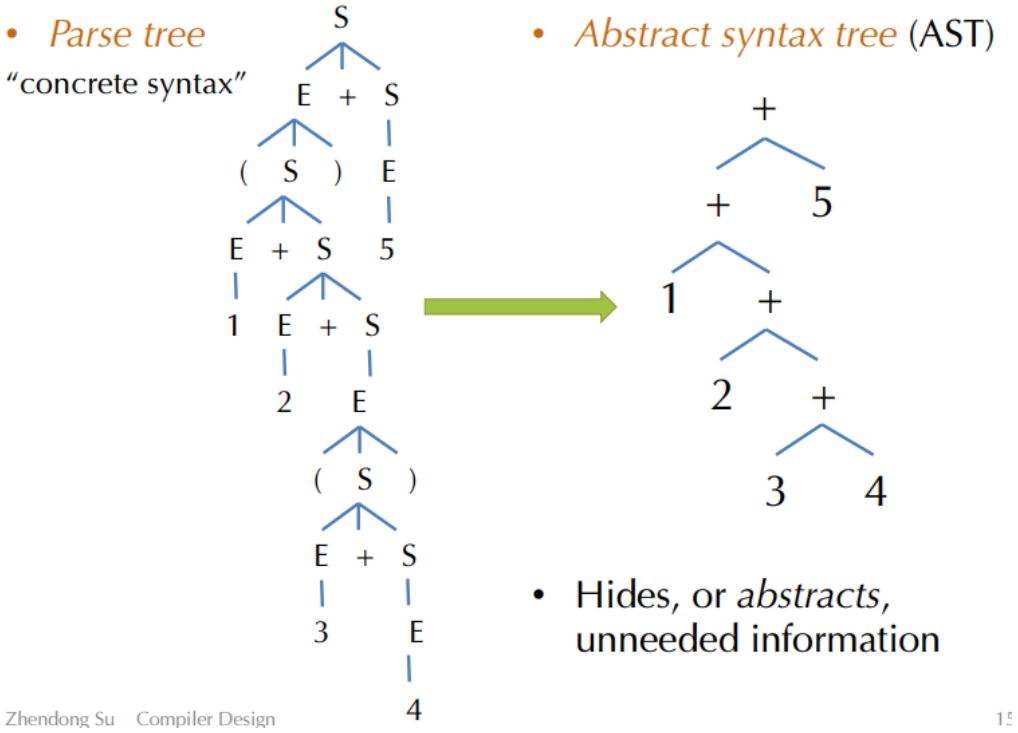
- The vertical | is shorthand for multiple productions
- 4 productions; 2 nonterminals - S, E; 4 terminals - (,), +, number; Start symbol - S

- Derivations in CFGs
  - For arbitrary strings α, β, γ and production rule: $A \mapsto \beta$, a single step of the derivation is

    $\alpha A \gamma \mapsto \alpha \beta \gamma$

- From derivations to Parse trees
  - Leaves: terminals

    In-order traversal yields the input token sequence
  - Internal nodes: non-terminals
  - No information on the order of the derivation steps
- From Parse Trees to Abstract Syntax



- Derivation Orders
  - Productions of the grammar can be applied in any order
  - Two standard orders

    Leftmost derivation - Find the left-most nonterminal and apply a production

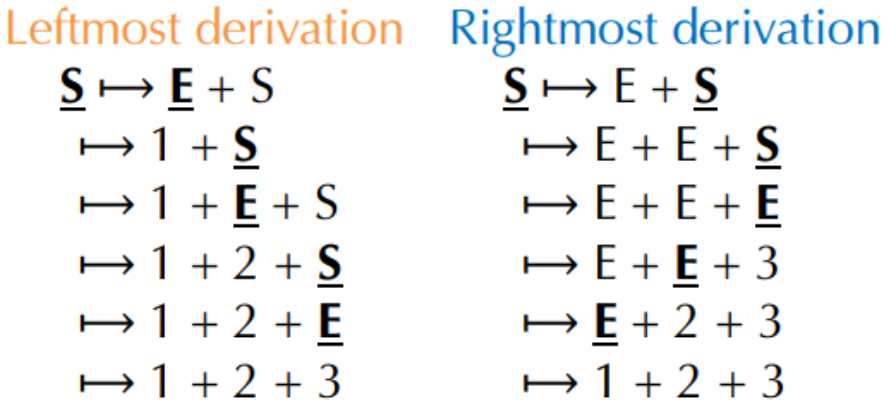
Rightmost derivation - ~ right-most ~

- Both strategies yield the same parse tree

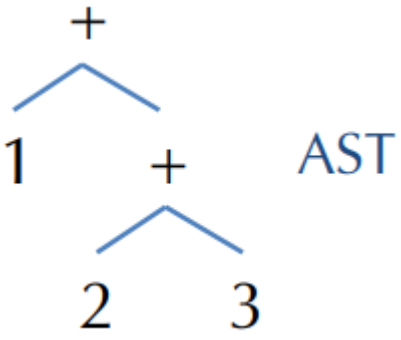
- Loops and Terminatoin
  - Consider $S \mapsto E$, $E \mapsto S$
  - or $S \mapsto (S)$
  - Easily generalize these examples to a "chain" of many nonterminals, which can be harder to find in a large grammar
  - Upshot: be aware of "vacuously empty" grammars

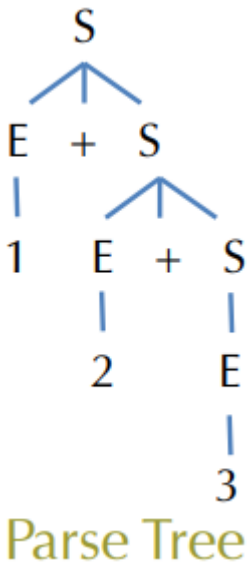
    Every nonterminal should eventually rewrite to an alternative that contains only terminal symbols

- Grammars for Programming Languages
  - Associativity
    - Consider the input: $1 + 2 + 3$, then we have both leftmost derivation and rightmost derivation:



    - This gives the AST and parse tree to be:





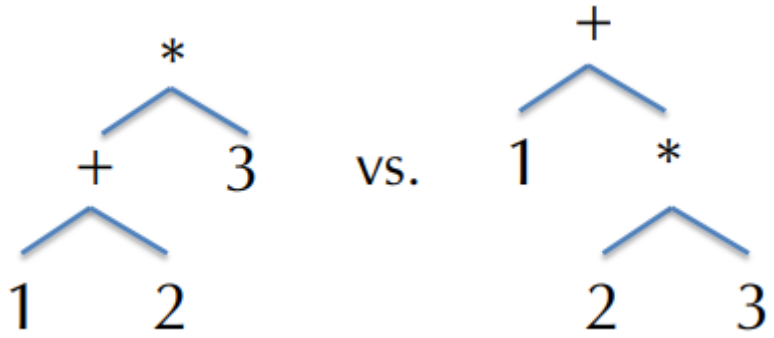
Parse Tree

- We normally believe + is left associative, but this grammar makes + right associative; this grammar is right recursive
- To make + left recursive, we make $S \mapsto S + E \mid E$

- Ambiguity
  - Consider $S \mapsto S + S \mid (S) \mid$ number.
  - Consider two leftmost derivations:

$$\underline{S} \mapsto \underline{S} + S \mapsto \underline{S} + S + S \mapsto 1 + \underline{S} + S \mapsto 1 + 2 + \underline{S} \mapsto 1 + 2 + 3$$
$$\underline{S} \mapsto \underline{S} + S \mapsto 1 + \underline{S} \mapsto 1 + \underline{S} + S \mapsto 1 + 2 + \underline{S} \mapsto 1 + 2 + 3$$

  - The first derivation gives left associativity; the second derivation gives right associativity
  - The same string/expression has two ASTs
- Why do we care about ambiguity?
  - Some operations are non-associative, some operations are only left/right associative
    - Right associative: assignment, exponentiation
    - Left associative: division, subtraction, modulo
    - Non-associative: $a > b > c$
  - Moreover, if there are multiple operations, ambiguity in the grammar leads to ambiguity in their precedence
  - Consider $S \mapsto S + S \mid S * S \mid (S) \mid$ number:



- Eliminating Ambiguity
    - Often eliminate ambiguity by adding nonterminals and allowing recursion only on the left (or right)
    - Higher-precedence operators go farther from the start symbol
    - To disambiguate
        - Decide to make $*$ higher precedence than $+$
        - Make $+$ left associative
        - Make $*$ right associative
    - $S_0 \mapsto S_0 + S_1 \mid S_1$

      $S_1 \mapsto S_2 * S_1 \mid S_2$

      $S_2 \mapsto \text{number} \mid (S_0)$
- Summary
    - CFGs allow concise specifications of programming languages
        - an unambiguous CFG specifies how to parse (convert a token stream to a parse tree)
        - ambiguity can be removed by encoding precedence and associativity in the grammar
    - Even with an unambiguous CFG, there may be more than one derivation - though all derivations correspond to the same AST
    - How to find a derivation?

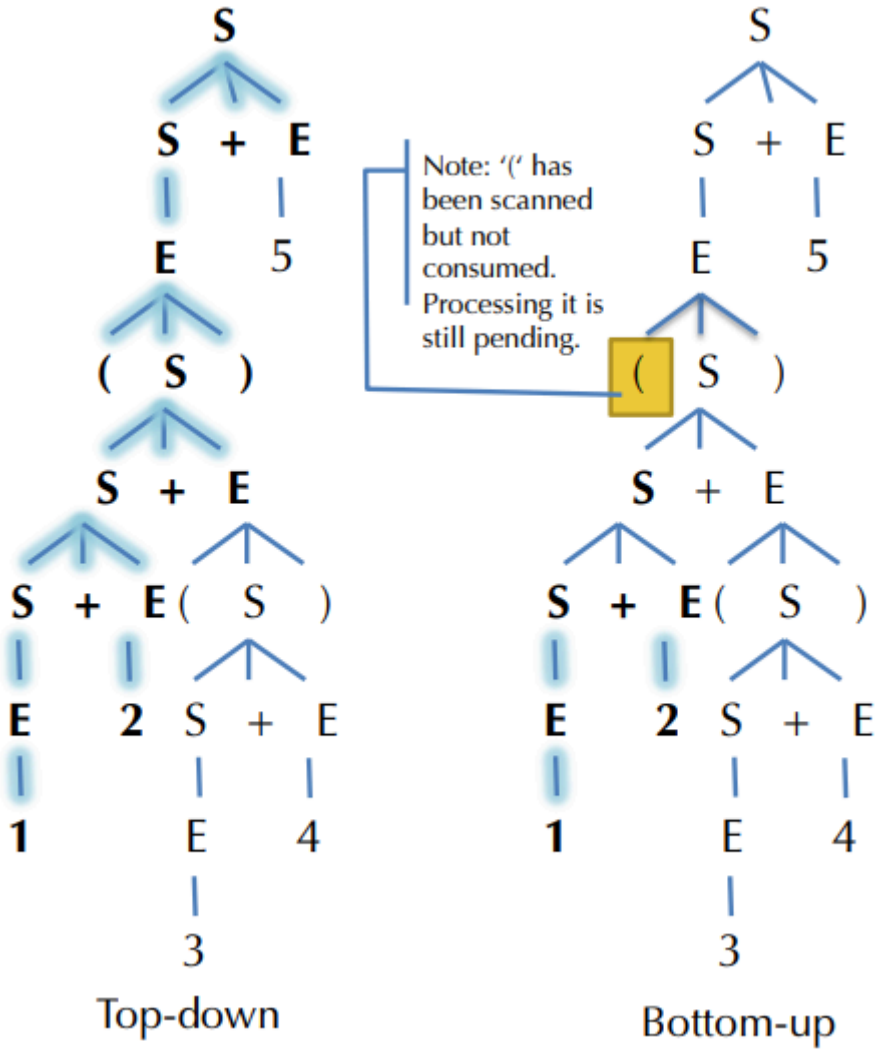
Oct. 21st - Exercise Session 3: LLVM

Oct. 23rd - Lecture 11: Top-down Parsing

- Check Goodnotes Temp document

Oct. 25th - Lecture 12: Bottom-up Parsing

- Recap
    - LL(1) Parsing
        - Scan from left to right; build leftmost derivation; one lookahead symbol at a time
        - Top-down parsing that finds the leftmost derivation
        - Language grammar → LL(1) grammar → prediction table → recursive-descent parser
        - Parser generator based on LL: ANTLR
        - Problems: grammar must be LL(1), Can extend to LL(k), grammar cannot be left recursive
- LR Grammars
    - Bottom-up Parsing (LR parsers)
        - LR(k) parser
            - Left to right scanning
            - rightmost derivation
            - k lookahead symbols
        - LR grammars are **more expressive** than LL
            - can handle left-recursive & right recursive grammars (virtually all programming languages)
            - easier to express programming language syntax (e.g. no left factoring)
        - Technique: "Shift-Reduce" parsers
            - Work bottom up instead of top down
            - Construct right-most derivation of a program in the grammar
            - Used by many parser generators (e.g. yacc, CUP, ocamlyacc, menhir, etc)
            - Better error detection/recovery
            - Poor error reporting (GCC's shift from bottom-up to top-down parsing)
    - Top-down vs. Bottom-up



Reductions	Scanned	Input Remaining
(1 + 2 + (3 + 4)) + 5 ⟵		(1 + 2 + (3 + 4)) + 5
(**E** + 2 + (3 + 4)) + 5 ⟵	(1	+ 2 + (3 + 4)) + 5
(**S** + 2 + (3 + 4)) + 5 ⟵	(1	+ 2 + (3 + 4)) + 5
(S + **E** + (3 + 4)) + 5 ⟵	(1 + 2	+ (3 + 4)) + 5
(**S** + (3 + 4)) + 5 ⟵	(1 + 2	+ (3 + 4)) + 5
(S + (**E** + 4)) + 5 ⟵	(1 + 2 + (3	+ 4)) + 5
(S + (**S** + 4)) + 5 ⟵	(1 + 2 + (3	+ 4)) + 5
(S + (S + **E**)) + 5 ⟵	(1 + 2 + (3 + 4	)) + 5
(S + (**S**)) + 5 ⟵	(1 + 2 + (3 + 4	)) + 5
(S + **E**) + 5 ⟵	(1 + 2 + (3 + 4)	) + 5
(**S**) + 5 ⟵	(1 + 2 + (3 + 4)	) + 5
E + 5 ⟵	(1 + 2 + (3 + 4))	+ 5
S + 5 ⟵	(1 + 2 + (3 + 4))	+ 5
S + **E** ⟵	(1 + 2 + (3 + 4)) + 5	
S		

Rightmost derivation

$S \longmapsto S + E \mid E$
$E \longmapsto \text{number} \mid (\ S\ )$

Zhendong Su Compiler Design

- Shift/Reduce Parsing
  - Parser state
    - stack of terminals and non-terminals
    - unconsumed input is a string of terminals

        - current derivation step is **stack + input**
    - Parsing is a sequence of **shift** and **reduce** operations
        - Shift: move look-ahead token to the stack
        - Reduce: replace symbols γ at top of stack with non-terminal s.t. $X \mapsto \gamma$ is a production, i.e. `pop` γ, `push` X
    -

Stack	Input	Action
	(1 + 2 + (3 + 4)) + 5	shift (
(	1 + 2 + (3 + 4)) + 5	shift 1
(1	+ 2 + (3 + 4)) + 5	reduce: E ⟼ number
(E	+ 2 + (3 + 4)) + 5	reduce: S ⟼ E
(S	+ 2 + (3 + 4)) + 5	shift +
(S +	2 + (3 + 4)) + 5	shift 2
(S + 2	+ (3 + 4)) + 5	reduce: E ⟼ number

- LR(0) Grammars
    - LR Parser States
        - Goal: know what set of reductions are legal at any given point
        - Idea: summarize all possible stack prefixes α as a finite parser state
            - Parser state is computed by a DFA that reads the stack σ
            - Accept states of the DFA correspond to unique reductions that apply
    - Example LR(0) Grammar: Tuples
        - $S \mapsto (L) \mid \text{id}$; $L \mapsto S \mid L, S$
        - Consider the string $(x, (y, z), w)$:
            - **Shift**: Move look-ahead token to the stack

Stack	Input	Action
	(x, (y, z), w)	shift (
(	x, (y, z), w)	shift x

            - **Reduce**: Replace symbols γ at top of stack with nonterminal X s.t. X ⟼ γ is a production, i.e., pop γ, push X

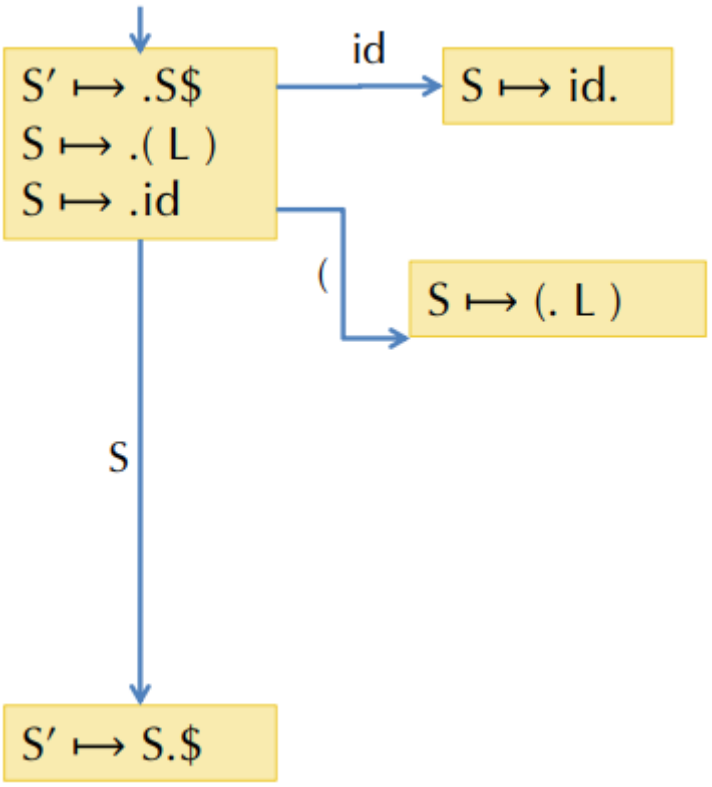
Stack	Input	Action
(x	, (y, z), w)	reduce S ⟼ id
(S	, (y, z), w)	reduce L ⟼ S

    - Action Selection Problem
        - Given a stack σ and a look-ahead symbol b, should the parser
            - shift b onto the stack (σb), or
            - reduce a production $X \mapsto \gamma$, assuming that $\sigma = \alpha\gamma$, and the new stack is αX

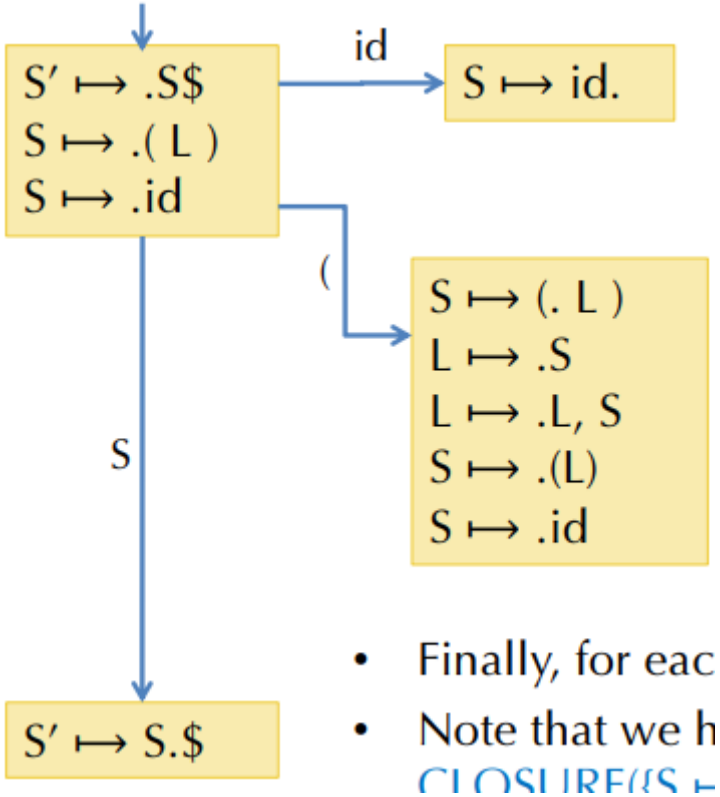
    - Sometimes, the parser can reduce but should not
    - Sometimes, the stack can be reduced in different ways
    - Main idea: Decide based on a prefix α of the stack plus look-ahed
    - Main goal: know what set of reductions are legal at any point
- LR(0) States
    - state: items to track progress on possible upcoming reductions
    - item: a production with an extra separator . in the RHS
    - e.g. $S \mapsto .(L)$, $S \mapsto (.L)$, $L \mapsto S.$
    - Intuition
        - Stuff before . is already on the stack (beginnings of possible γs to be reduced)
        - Stuff after the . is what might be seen next
        - The prefixes α are represented by the state itself
- Constructing the DFA: Start state and closure
    - First step: add a new production $S' \mapsto S\$$ to the grammar
    - Start state of the DFA = empty stack, so it contains the item $S' \mapsto .S\$$
    - Closure of a state
        - Adds items for all productions whose LHS nonterminal occurs in an item in the state just after the .
        - The added items have the . located at the beginning (no symbols for those items have been added to the stack yet)
        - Note that newly added items may cause yet more items to be added to the state... keep iterating until a fixed point is reached

        CLOSURE({ S' ⟼ .S$ }) = { S' ⟼ .S$, S ⟼ .(L), S⟼.id }

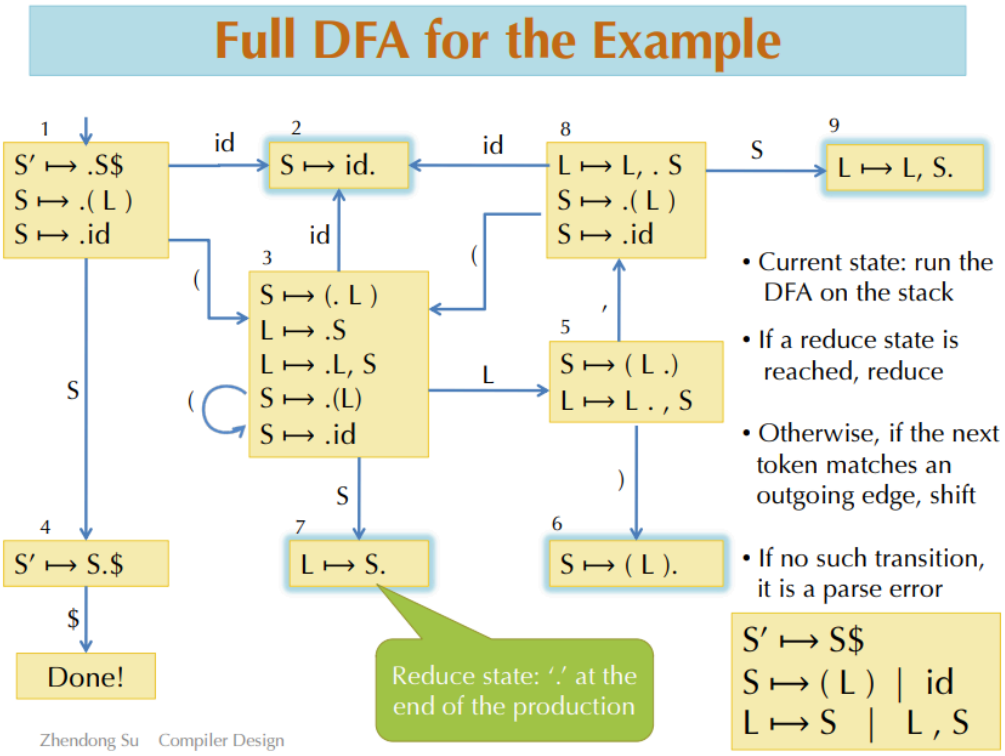
        - Resulting "closed state" contains the set of all possible productions that might be reduced next
    - Next, we take the closure that state
    - In the set of items, the nonterminal S appears after the ., so we add items for each S production in the grammar



- Next we add the transitions



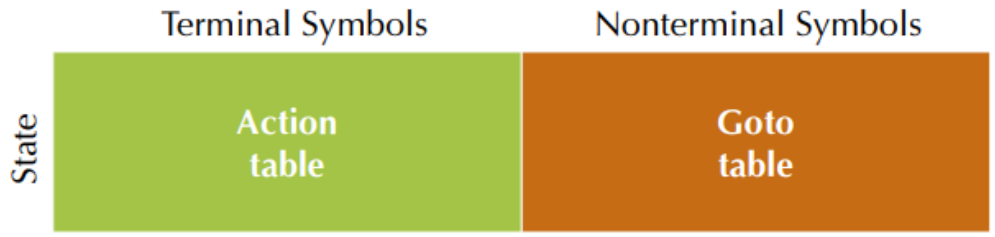
- Finally for each new state, we take the closure



- Reduce state: . at the end of the production

- Using the DFA
  - Run parser through the DFA
  - Resulting state tells what productions may be reduced next
    - If not in a reduce state, shift the next symbol & transition w.r.t. DFA
    - If in a reduce state, $X \mapsto \gamma$ with stack $\alpha\gamma$, pop γ, and push X
  - Optimization: no need to rerun DFA from beginning each step
    - Store the state with each symbol on the stack
    - On a reduction $X \mapsto \gamma$, pop stack to reveal the state too
    - Next, push the reduction symbol to reach stack
    - Then take just one step in the DFA to find next state
- Implementing the Parsing Table
  - Represent the DFA as a table of shape: state * (terminals + nonterminals)
  - Entries for the "action table" specify two kinds of actions
    - Shift and go to state n
    - Reduce using reduction $X \mapsto \gamma$

- First pop γ off the stack to reveal the state, look up X in the "goto table" and go to that table



- Example Parse Table

	(	)	id	,	$	S	L
1	s3		s2			g4	
2	S↦id	S↦id	S↦id	S↦id	S↦id		
3	s3		s2			g7	g5
4					DONE		
5		s6		s8			
6	S ↦ (L)	S ↦ (L)	S ↦ (L)	S ↦ (L)	S ↦ (L)		
7	L ↦ S	L ↦ S	L ↦ S	L ↦ S	L ↦ S		
8	s3		s2			g9	
9	L ↦ L,S	L ↦ L,S	L ↦ L,S	L ↦ L,S	L ↦ L,S		

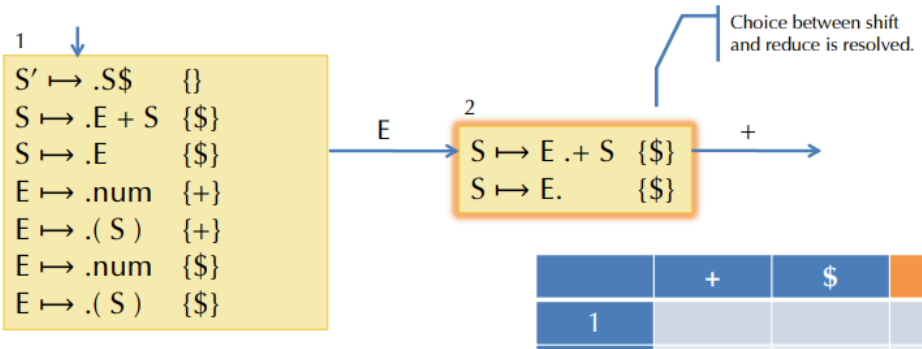
sx = shift and go to state x
gx = go to state x

- LR(0) limitations
    - An LR(0) machine only works if states with reduce actions have a single reduce action
        - In such states, the machine always reduces (ignoring lookahead)
    - With more complex grammars, the DFA construction will yield states with **shift/reduce** and **reduce/reduce** conflicts
    - Such conflicts can often be resolved using a single lookahead
- LR(1) Parsing
    - Algorithm is similar to LR(0) DFA construction
        - LR(1) state = set of LR(1) items
        - An LR(1) item is an LR(0) item + a set of look-ahea symbols
        - $A \mapsto \alpha.\beta, \mathscr{L}$
    - LR(1) closure is more complex
    - Form the set of items just as for LR(0) algorithm

- Whenever a new item $C \mapsto .\gamma$ is added because $A \mapsto \beta.C\delta$, $\mathscr{L}$ is already in the set, we need to compute its look-ahead set $\mathscr{M}$
    - The look-ahead set $\mathscr{M}$ includes FIRST(δ), the set of terminals that may start strings derived from δ
    - If δ is or can derive ε, then the look-ahead $\mathscr{M}$ also contains $\mathscr{L}$
- Example Closure

S′ ⟼ S$
S ⟼ E + S | E
E ⟼ number | (S)

    - Start item: S′ ⟼ .S$, {}
    - Since S is to the right of a '.', add
      S ⟼ .E + S , {$} Note: {$} is FIRST($)
      S ⟼ .E , {$}
    - Need to keep closing, since E appears to the right of a '.' in '.E + S'
      E ⟼ .number , {+} Note: + added for reason 1
      E ⟼ .(S) , {+} FIRST(+ S) = {+}
    - Because E also appears to the right of '.' in '.E' we get:
      E ⟼ .number , {$} Note: $ added for reason 2
      E ⟼ .(S) , {$} δ is ε
    - All items are distinct, so we're done
- Using the DFA

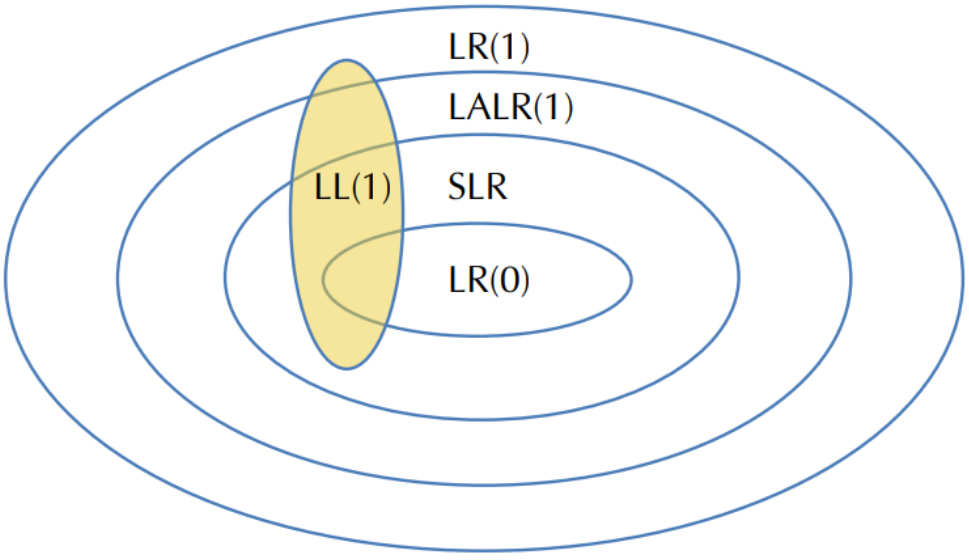


	+	$	E
1			g2
2	s3	S ⟼ E	

Fragment of the Action & Goto tables

    - The behavior is determined if
        - there is no overlap among the look ahead sets for each reduce item, and
        - none of the look-ahead symbols appear to the right of a .
- LR(1) issues

        - LR(1) gives maximal power out of a 1 look-ahead symbol parsing table
            - DFA + stack is a push-down automaton
        - In practice, LR(1) tables are big
            - modern implementations directly generate code
- LR variants: LALR(1) & GLR
        - LALR(1) states
            - stands for LookAhead LR
            - Typically 10 times fewer states than LR(1)
            - Merge the LR(1) states with the same core, but different lookahead
        - May introduce new reduce/reduce conflicts, but not new shift/reduce conflicts
        - SLR(1) = "Simple" LR (like LR(0), but use the FOLLOW information)
        - GLR = "Generalized LR" parsing
            - efficiently compute the set of all parses for a given input
            - later passes should disambiguate based on other context
- Classification of Grammars



Oct. 30th - Lecture 13: First Class Functions I

- Menhir in Practice
    - Practical Issues
        - Dealing with source file location information
            - In the lexer and parser
            - In the abstract syntax
        - Lexing comments/strings
    - Precedence and associativity declarations
        - parser generators often support precedence/associativity declarations
            - hints to the parser about how to resolve conflicts
        - Pros
            - avoids having to manually resolve those ambiguities by manually introducing extra nonterminals
            - easier to maintain the grammar
        - Cons
            - Cannot as easily re-use the same terminal
            - introduces another level of debugging
        - Limits
            - not always easy to disambiguate just with precedence/associativity
    - Example ambiguity in real languages
        - Consider the grammar

          $S \mapsto$ `if (E) S`

          $S \mapsto$ `if (E) S else S`

          $S \mapsto X = E$, ...
        - Consider how to parse

          `if (E1) if (E2) S1 else S2`
            - "dangling else" problem
    - How to disambiguate `if-then-else`
        - Want to rule out: `if (E1) {if (E2) S1} else S2`
        - Observation: an un-matched "if" should not appear as the "then" clause of a containing "if"

$S \mapsto M \mid U$ (M = "matched", U = "unmatched")

$U \mapsto$ `if (E) S` (Unmatched "if")

$U \mapsto$ `if (E) M else U` (Nested if is matched)

$M \mapsto$ `if (E) M else M` (Matched "if")

$M \mapsto X = E$ (Other statements)

- We can add braces, require indentation, must match "if-else" from the programmer

- Alternative: Use {}
  - Ambiguity arises because the "then" branch is not well bracketed
  - One could just require brackets
    - requiring them for the else clause leads to ugly code for chained if-statements
  - Compromise
    - allow unbracketed else block only if the body is "if"

- OAT V1.0
  - Simple C-like imperative language
    - supports 64-bit integers, arrays, strings
    - top-level, mutually recursive procedures
    - scoped local, imperative variables
  - Design/specify?
    - Grammatical constructs
    - Semantic constructs
  - Static semantics vs. Dynamic semantics
- First-class functions (untyped lambda calculus, substitution, evaluation)
  - "Functional" languages
    - languages like ML, Haskell, Scheme, Python, C#, Java 8, Swift
      - Functions can be passed as arguments
      - Functions can be returned as values
      - Function nest: inner function can refer to variables bound in the outer function
  - (Untyped) Lambda Calculus
    - The lambda calculus is a minimal programming language

      `(fun x -> e)`, lambda-calculus notation: $\lambda x.e$

        - It has variables, functions, and function application
            - It's Turing Complete: it either supports loops or supports recursion
            - Basis and foundation for other programming languages
- Values and Substitution
    - The only values of the lambda calculus are (closed) functions

```
val ::=
    | fun x -> exp
```

    - To **substitute** value `v` for variable `x` in expression `e`
        - Replace all **free occurrences** of `x` in `e` by `v`
        - In OCaml: written `subst v x e`
        - In math: written $e\{v/x\}$
    - Function application is interpreted by substitution

```
(fun x -> fun y -> x + y) 1
subst 1 x (fun y -> x + y)
fun y -> 1 + y
```

- Lambda Calculus Operational Semantics
    - Substitution function (in Math)

```
x{v/x}              = v
// replace the free x by v
y{v/x}              = y
// assume y != x
(fun x -> exp){v/x} = (fun x -> exp)
// x is bound in exp
(fun y -> exp){v/x} = (fun y -> exp{v/x})
// assume y != x
(e1 e2){v/x}        = (e1{v/x} e2{v/x})
// substitute everywhere
```

    - Examples
        - x y {(fun z -> z)/y} → x (fun z -> z)
- Free variables and scoping

- ```
  let add = fun x -> fun y -> x + y
  let inc = add 1
  ```

    - The result of `add 1` is a function
    - After calling `add` we cannot throw away its argument because those are needed in the function returned by `add`
    - We say variable `x` is **free** in `fun y -> x + y`
        - free variables are defined in an outer scope
    - We say variable `y` is bound by `fun y`
        - Its scope is the body "`x + y`" in `fun y -> x + y`
    - A term with no free variables is called **closed**
    - A term with one or more free variables is called **open**
- Free Variable Calculation
    - OCaml code to compute the set of free variables in lambda expressions

      ```
      let rec free_vars (e:exp) : VarSet.t =
        begin match e with
          | Var x        -> VarSet.singleton x
          | Fun(x, body) -> VarSet.remove x (free_vars body)
          | App(e1, e2)  -> VarSet.union (free_vars e1) (free_vars e2)
        end
      ```

    - Lambda expression `e` is closed if `free_vars e` is `VarSet.empty`
- Variable Capture
    - Note that if we try to naively "substitute" an open term, a bound variable might **capture** the free variables

      (fun x -> (x y)){(fun z -> x)/y}

      fun x -> (x (fun z -> x))
        - free `x` is **captured**
    - Usually **not** the desired behaviour
        - this property is sometimes called "dynamic scoping"
        - The meaning of "x" is determined by where it is bound dynamically (not where it is bound statically)
        - some languages are implemented with this as a "feature"
        - But, leads to hard to debug scoping issues
- Alpha equivalence
      ```

        - Two terms that differ only by consistent renaming of bound variables are called **alpha equivalent**
        - The names of free variables do matter
    - Fixing substitution
        - Consider $e_1\{e_2/x\}$
            - to avoid capture, define substitution to pick an alpha equivalent version of e_1 such that the bound names of e_1 don't mention the free names of e_2
            - Then do the naive substitution
    - Operational semantics
        - denotation semantics, operational semantics, axiomatic semantics
        - Specified with 2 inference rules with judgements of the form $\texttt{exp} \downarrow v$
            - Read this notation as "program `exp` evaluates to value v"
            - We give a **call-by-value** semantics (function arguments are evaluated before substitution)
        - Rules

$$\frac{}{v \Downarrow v}$$

"Values evaluate to themselves"

$$\frac{exp_1 \Downarrow (fun\ x \rightarrow exp_3) \qquad exp_2 \Downarrow v \qquad exp_3\{v/x\} \Downarrow w}{exp_1\ exp_2 \Downarrow w}$$

"To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function. "

    - The omega term - an infinite loop written in untyped lambda calculus

      `(fun x -> x x) (fun x -> x x)`

      $\rightarrow$ `(x x){(fun x -> x x)/x}`

      $=$ `(fun x -> x x) (fun x -> x x)`

- Implementing the interpreter
    - Y Combinator & Factorial
        - It calculates fixpoints of a function f
        - $g(\texttt{Y}\ g) = \texttt{Y}\ g$
        - $Y = \lambda f.(\lambda x.\, f(\texttt{x}\ \texttt{x}))(\lambda x.\, f(\texttt{x}\ \texttt{x}))$

- Defining the factorial function
    - Typical recursive definition

      `fac` $= \lambda$`n. if (n = 0) 1 (n * fac(n-1))`
    - Abstract it: $F = \lambda$`f.` λ`n. if (n = 0) 1 (n * fac(n-1))`
    - Thus, `fac` $= F$ `fac`
    - $Y\ F$ being fixpoint of F can thus be viewed as the factorial function

Nov. 1st - Lecture 14: First Class Functions II

- Implementing the Interpreter
    - Adding Integers to Lambda Calculus

exp ::=	
| …	
| **n**	*constant integers*
| **exp_1 + exp_2**	*binary arithmetic operation*
val ::=	
| `fun x -> exp`	*functions are values*
| **n**	*integers are values*
n{v/x} = **n**	*constants have no free vars*
$(e_1 + e_2)${v/x} = **(e_1{v/x} + e_2{v/x})**	*substitute everywhere*

$$\frac{exp_1 \Downarrow n_1 \quad exp2 \Downarrow n_2}{exp_1 + exp_2 \Downarrow (n1\ [\![+]\!]\ n2)}$$

object-level '+' meta-level '+'

- Static analysis (Scope, Types, and Context)
    - Variable Scoping
        - Problem: how to determine whether a declared variable is in scope?
        - Issues
            - Which variables are available at a given program location?
            - Can the same identifier be reused (i.e. shadowing), or it is an error?

- Code below is syntactically correct, but not well-formed

```
int fact(int x) {
    var acc = 1;
    while (x > 0) {
        acc = acc * y;
        x = q - 1;
    }
    return acc;
}
```

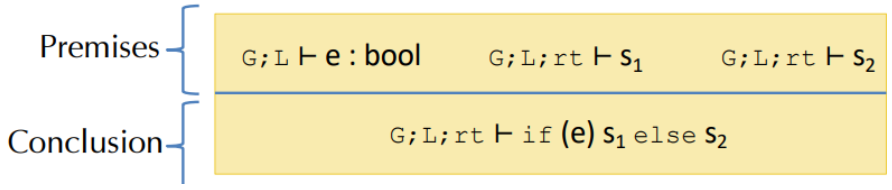
    - `y, q` are used without being defined anywhere
  - Can we solve the problem by changing the parser to rule out such programs?
    - Parser not designed to solve this type of problem.
- Contexts and Inference Rules
  - Need to track of contextual information
    - what variables are in scope
    - what are their types
  - How to describe this
    - compiler keeps a mapping from variables to information about them
    - using a "**symbol table**"
- Why inference rules
  - Allow a compact, precise way of specifying language properties
    - about 20 pages for full Java, vs
    - 100+ pages of prose Java Language Specification
  - Correspond closely to recursive AST traversal for implementing them
  - Type checking / inference tries to prove a different judgement `G; L ⊢ e: t`
    - By searching backward through the rules
  - Compiling is also a set of inference rules specifying `G ⊢` source → target
    - compilation judgements are similar to the type checking judgements

        - Strong mathematical foundations (Curry-Howard Correspondence)
            - Programming language ~ Logic
            - Program ~ Proof
            - Type ~ Proposition
- Inference Rules
    - A judgement `G; L ⊢ e: t` is read "the expression `e` is well typed and has type `t`"
    - For any environment `G; L`, expression `e`, and statements `s1, s2` all,

$$G;L;rt \vdash \texttt{if (e) } s_1 \texttt{ else } s_2$$

holds if $G;L \vdash e : bool$, $G;L;rt \vdash s_1$, $G;L;rt \vdash s_2$ all hold

    - More succinctly, summarize these constraints as an **inference rule**



    - It can be used for **any substitution** of the metavariables `G, L, rt, e, s1, s2`
- Checking derivations
    - Derivation or proof tree
        - nodes: judgements
        - edges: connect premises to a conclusion (according to inference rules)
    - Leaves of the tree are **axioms** (rules with no premises)
    - Goal of the type checker: verify that such a tree exists
- Compilation as Translating Judgements
    - Consider the typing judgement for source expressions $C \vdash e : t$
    - How to interpret this info in the target language, that is $[C \vdash e : t]$ =?
        - $[t]$ is a target type
        - $[e]$ translates to a (possibly empty) sequence of instructions (the instruction sequence compute e's result into some operand)
    - Invariant

- If $[C \vdash e : t] = \texttt{ty, operand, stream}$, then the type (at the target level) of the operand is $\texttt{ty} = [t]$

- Example for compilation as translating judgements

C ⊢ 341 + 5 : int what is ⟦ C ⊢ 341 + 5 : int⟧?

⟦ ⊢ 341 : int ⟧ = (i64, Const 341, []) ⟦⊢ 5 : int⟧ = (i64, Const 5, [])

⟦C ⊢ 341 : int⟧ = (i64, Const 341, []) ⟦C ⊢ 5 : int⟧ = (i64, Const 5, [])

⟦C ⊢ 341 + 5 : int⟧ = (i64, %tmp, [%tmp = add i64 (Const 341) (Const 5)])

- What about the Context?
  - Source level C has bindings like: $x : \texttt{int}, y : \texttt{bool}$
    - think of it as a finite map from identifiers to tpes
  - $[C]$ maps source identifiers x to source types and $[x]$
  - Interpretation of a variable $[x]$ at the target level

$$\frac{x{:}t \in L}{G;L \vdash x : t}\ \textbf{TYP\_VAR}$$
as expressions
(which denote values)

$$\frac{x{:}t \in L \quad G;L \vdash exp : t}{G;L;rt \vdash x = exp; \Rightarrow L}\ \textbf{TYP\_ASSN}$$
as addresses
(which can be assigned)

- Interpretation of Contexts
  - $[C]$ = a map from source identifiers to **types** and **target identifiers**
  - Invariant $x : t \in C$ means that
    - lookup $[C]x = (t, \texttt{\%id\_x})$
    - the target type of $\texttt{\%id\_x}$ is $[t]^*$ (a pointer to $[t]$)

- Interpretation of Variables
  - Establish invariant for expressions

$$\left[\!\!\left[\frac{x{:}t \in L}{G;L \vdash x : t}\ \textbf{TYP\_VAR} \right]\!\!\right] = \texttt{(\%tmp, [\%tmp = load i64* \%id\_x])}$$
as expressions
(which denote values)

where (i64, %id_x) = lookup ⟦L⟧ x

  - Invariant for statements

$$\left[\!\left[\frac{x{:}t \in L \quad G;L \vdash exp : t}{G;L;rt \vdash x = exp; \Rightarrow L}\ \text{TYP\_ASSN}\right]\!\right] = \text{stream @ } \texttt{[store [[t]] opn, [[t]]* \%id\_x]}$$

as addresses (which can be assigned)

where (t, %id_x) = lookup [[L]] x
and [[G;L ⊢ exp : t]] = ([[t]], opn, stream)

- Other Judgements?
  - Statement
    [[C; rt ⊢ stmt ⇒ C']] = [[C']] , stream
  - Declaration
    [[G;L ⊢ t x = exp ⇒ G;L,x:t]] = [[G;L,x:t]], stream

    Invariant: stream is of the form

    ```
    stream' @
    [ %id_x = alloca [[t]];
      store [[t]] opn, [[t]]* %id_x ]
    ```

    and [[G;L ⊢ exp : t]] = ([[t]], opn, stream')
- Compiling Control
  - Translating `while`
    - Consider translating `while(e) s`
      - test conditional `e`, if true jump to body `s`, else jump to label after body `s`
    - $[C; \texttt{rt} \vdash \texttt{while(e) s} \rightarrow C'] = [C']$

      ```
      lpre:
          opn = [C vdash e : bool]
          %test = icmp eq i1 opn, 0
          br %test, label %lpost, label %lbody
      lbody:
          [C;rt vdash s to C']
          br %lpre
      lpost:
      ```

    - writing `opn` $= [C \vdash e : \texttt{bool}]$ is pun
      - translating it generates code that puts the result into `opn`
  - Translating `if-then-else`

$$[\![C; rt \vdash \texttt{if } (e_1)\ s_1 \texttt{ else } s_2 \Rightarrow C']\!] = [\![C']\!],$$

```
    opn = [[C ⊢ e : bool]]
    %test = icmp eq i1 opn, 0
    br %test, label %else, label %then
then:
    [[C;rt ⊢ s1 ⇒ C']]
    br %merge
else:
    [[C;rt ⊢ s2 ⇒ C']]
    br %merge
merge:
```

- Connecting this to code
    - instruction streams
        - must include labels, terminators, and "hoisted" global constants
    - must post-process the stream into a control-flow-graph
- Optimizing Control
    - Standard evaluation

```
if (x & !y | !w)
  z = 3;
else
  z = 4;
return z;
```

```
    %tmp1 = icmp Eq [[y]], 0      ; !y
    %tmp2 = and [[x]] [[tmp1]]
    %tmp3 = icmp Eq [[w]], 0
    %tmp4 = or %tmp2, %tmp3
    %tmp5 = icmp Eq %tmp4, 0
    br %tmp4, label %else, label %then

then:
    store [[z]], 3
    br %merge

else:
    store [[z]], 4
    br %merge

merge:
    %tmp5 = load [[z]]
    ret %tmp5
```


- Observation
    - Usually, we want the translation $[e]$ to produce a value
        - $[C \vdash e : t]$ = `(ty, operand, stream)`
    - But when the expression we are compiling appears in a test, the program jumps to one label or another after the comparison but otherwise never uses the value
    - In many cases, we can avoid "materializing" the value and thus produce better code

- Idea: use a different translation for tests
    - Conditional branch translation of booleans, without materializing value

      $[C \vdash e : \texttt{bool@}]\ \texttt{ltrue lfalse} = \texttt{stream}$
    - Two extra arguments: "true" branch label, "false" branch label
    - Does not return a value

$[\![C, rt \vdash \texttt{if (e) then s1 else s2} \Rightarrow C']\!] = [\![C']\!],$

```
    insns3
then:
    [[s1]]
    br %merge
else:
    [[s2]]
    br %merge
merge:
```

where

$[\![C, rt \vdash s_1 \Rightarrow C']\!] = [\![C']\!], insns_1$

$[\![C, rt \vdash s_2 \Rightarrow C']\!] = [\![C']\!], insns_2$

$[\![C \vdash e : bool@\ ]\!]\ \texttt{then else} = insns_3$

- Short Circuit Compilation: expressions

$[\![C \vdash e : bool@]\!]\ ltrue\ lfalse = insns$

$$\frac{}{[\![C \vdash false : bool@]\!]\ ltrue\ lfalse = [\texttt{br \%lfalse}]}\ \text{FALSE}$$

$$\frac{}{[\![C \vdash true : bool@]\!]\ ltrue\ lfalse = [\texttt{br \%ltrue}]}\ \text{TRUE}$$

$$\frac{[\![C \vdash e : bool@]\!]\ lfalse\ ltrue = insns}{[\![C \vdash !e : bool@]\!]\ ltrue\ lfalse = insns}\ \text{NOT}$$

- Short Circuit Evaluation - Idea: build the logic into the translation

$$\frac{[\![C \vdash e1 : bool@]\!]\ ltrue\ right = insns_1 \quad [\![C \vdash e2 : bool@]\!]\ ltrue\ lfalse = insns_2}{[\![C \vdash e1 \mid e2 : bool@]\!]\ ltrue\ lfalse =}$$

```
    insns1
right:
    insn2
```

$$\frac{[\![C \vdash e1 : bool@]\!]\ right\ lfalse = insns_1 \quad [\![C \vdash e2 : bool@]\!]\ ltrue\ lfalse = insns_2}{[\![C \vdash e1 \,\&e2 : bool@]\!]\ ltrue\ lfalse =}$$

```
    insns1
right:
    insn2
```

where `right` is a fresh label

- Short Circuit Evaluation: shortening conditional program fragment

```
if (x & !y | !w)
  z = 3;
else
  z = 4;
return z;
```

```
    %tmp1 = icmp Eq [[x]], 0
    br %tmp1, label %right2, label %right1

right1:
    %tmp2 = icmp Eq [[y]], 0
    br %tmp2, label %then, label %right2

right2:
    %tmp3 = icmp Eq [[w]], 0
    br %tmp3, label %then, label %else

then:
    store [[z]], 3
    br %merge

else:
    store [[z]], 4
    br %merge

merge:
    %tmp5 = load [[z]]
    ret %tmp5
```

Zhendong Su Compiler Design

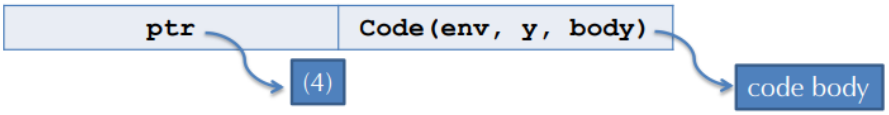
Nov. 4th - Exercise Session 4

Nov. 6th - Lecture 15: Closure Conversions and Types - Judgements and Derivations

- Closure Conversion (compiling lambda calculus to straight-line code, representing evaluation environments at runtime)
    - Compiling First-class functions
        - 2 problems
            - must implement substitution of free variables
            - must separate "code" from "data"

- Reify the substitution
    - move substitution from the meta language to the object language by making the data structure & lookup operation explicit
    - the environment-based interpreter is one step in this direction
- Closure conversion
    - eliminates free variables by packaging up the needed environment in the data structure
- Hoisting
    - separates code from data, pulling closed code to the top level

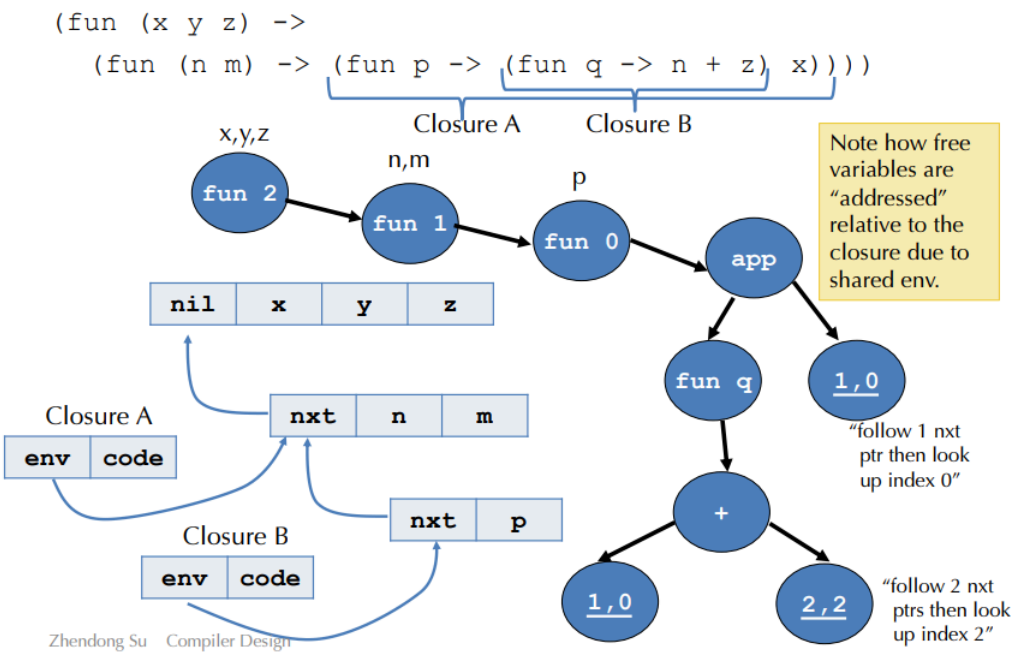
- Example of Closure Creation
    - Recall `add` function: `let add = fun x -> fun y -> x + y`
    - Consider inner function: `fun y -> x + y`
    - When run the function application: `add 4`, the program builds a closure and returns it
        - the closure is **a pair of the environment and a code pointer**



    - The code pointer takes a pair of parameters: `env` and `y`
        - the function code is essentially:
          `fun (env, y) -> let x = nth evn 0 in x + y`

- Representing Closures
    - Simple closure conversion does not generate very efficient code
        - It stores all the values for variables in the environment, even if they are not needed by the function body
        - It copies the environment values each time a nested closure is created
        - It uses a linked-list data structure for tuples
    - Many options
        - Store only the values for free variables in the body of the closure

        - Share subcomponents of the environment to avoid copying
        - Use vectors or arrays rather than linked structures
    - Array-based Closures with N-ary Functions



- Static Analysis
    - Adding Integers to Lambda Calculus
    - Variable Scoping
    - Type Checking / Static Analysis

The interpreter from the Eval3 module (fun.ml, Lec 13)

```
let rec eval env e =
  match e with
  | …
  | Add (e1, e2) ->
      (match (eval env e1, eval env e2) with
         | (IntV i1, IntV i2) -> IntV (i1 + i2)
         | _ -> failwith "tried to add non-integers")
  | …
```

        - The interpreter might fail at runtime
            - not all operations are defined for all values
            - e.g. $3/0$, $3 + \texttt{true}$
        - A compiler cannot generate sensible code for this case
            - a naive implementation might "add" an integer and a function pointer

- Statically Ruling Out Partiality: Type Checking
    - Note about this Typechecker

        - The interpreter only evaluates the body of a function when it's applied
        - Typechecker always check function's body (even if it's never applied)
            - assume the input has some type t_1
            - Reflect this information in the type of the function ($t_1 \to t_2$)
        - Dually, at a call site ($e_1\ e_2$), we do not know what **closure** we will get, but we can
            - calculate e_1's type
            - check e_2 is an argument of the right type
            - determine what type e_1 will return
        - This is an approximation
        - What if `well_typed` always returns `false`: it is a sound typechecker but useless
- Contexts and Inference Rules
    - Need to keep track of contextual information
        - what variables are in scope
        - what are their types
        - what information do we have about each syntactic construct
    - What relationships are there among the syntactic objects?
        - is one type a subtype of another?
    - How do we describe this information
        - in the compiler, there is a mapping from variables to information we know about them
        - the compiler has a collection of (mutually recursive) functions that follow the structure of the syntax
- Type judgements
    - In the judgement: $E \vdash e : t$
        - E is a **typing environment** or a **type context**
        - E maps variables to types and is simply a set of bindings of the form: $x_1 : t_1; x_2 : t_2; \ldots; x_m : t_m$
        - e.g. x : `int`, b : `bool` $\vdash$ `if (b) 3 else x` : int
- Simply-typed lambda calculus

$$\text{INT}\ \frac{}{E \vdash i : \text{int}} \qquad \text{VAR}\ \frac{x : T \in E}{E \vdash x : T} \qquad \text{ADD}\ \frac{E \vdash e_1 : \text{int} \quad E \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}}$$

$$\text{FUN}\ \frac{E, x : T \vdash e : S}{E \vdash \text{fun } (x{:}T) \text{ -> } e \ : T \text{ -> } S} \qquad \text{APP}\ \frac{E \vdash e_1 : T \text{ -> } S \quad E \vdash e_2 : T}{E \vdash e_1\ e_2 : S}$$

- Type checking derivations
    - Derivation or proof tree
    - Leaves of the tree are axioms
    - Goal of type checker: verify that such a tree exists
- Example Derivation Tree

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \text{int} \in x : \text{int}}\,\text{VAR}}{x : \text{int} \vdash x : \text{int}} \quad \dfrac{}{x : \text{int} \vdash 3 : \text{int}}\,\text{INT}}{x : \text{int} \vdash x + 3 : \text{int}}\,\text{ADD}}{\vdash (\text{fun } (x{:}\text{int}) \text{ -> } x + 3) : \text{int -> int}}\,\text{FUN} \quad \dfrac{}{\vdash 5 : \text{int}}\,\text{INT}}{\vdash (\text{fun } (x{:}\text{int}) \text{ -> } x + 3)\ 5 : \text{int}}\,\text{APP}$$

- Type Safety
    - Theorem (simply typed lambda calculus with integers): If $\vdash e : t$, then $\exists v, e \downarrow v$
    - Well-typed programs never executes undefined code like `3+(fun x -> 2)`
    - Simply-typed lambda calculus terminates
- Type Safety for General Languages
    - Theorem (Type Safety): If $\vdash P : t$ is a well-typed program then either (a) the program terminates in a well-defined way, or (b) the program continues computing forever
    - Well-defined termination could include
        - halting with a return value
        - raising an exception
- Type safety rules out undefined behaviour
    - abusing "unsafe" casts: converting pointers to integers

    - treating non-code values as code
    - breaking the type abstractions of the language
    - What is "defined" depends on the language semantics
- Arrays
    - First add a new type constructor: `T[]`

NEW
$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : T}{E \vdash \text{new } T[e_1](e_2) \ : T[]}$$

e_1: size of newly alloc. array
e_2: initializes the array

INDEX
$$\frac{E \vdash e_1 : T[] \qquad E \vdash e_2 : \text{int}}{E \vdash e_1[e_2] \ : T}$$

UPDATE
$$\frac{E \vdash e_1 : T[] \quad E \vdash e_2 : \text{int} \quad E \vdash e_3 : T}{E \vdash e_1[e_2] = e_3 \text{ ok}}$$

Note: These rules don't ensure array indices are within bounds, which should be checked *dynamically*

- Tuples
    - ML-style tuples with statically known number of products
    - First, add a new type constructor: $T_1 * \cdots * T_n$

TUPLE
$$\frac{E \vdash e_1 : T_1 \quad \ldots \quad E \vdash e_n : T_n}{E \vdash (e_1, \ldots, e_n) : T_1 * \ldots * T_n}$$

PROJ
$$\frac{E \vdash e : T_1 * \ldots * T_n \quad 1 \le i \le n}{E \vdash \#i\ e \ : \ T_i}$$

- References
    - ML-style references
    - First, add a new type constructor: `T ref`

REF

$$\frac{E \vdash e : T}{E \vdash \text{ref } e : T \text{ ref}}$$

DEREF

$$\frac{E \vdash e : T \text{ ref}}{E \vdash !e : T}$$

ASSIGN

$$\frac{E \vdash e_1 : T \text{ ref} \quad E \vdash e_2 : T}{E \vdash e_1 := e_2 : \text{unit}}$$

Note the similarity with the rules for arrays

- Types, More Generally
    - What are types, anyway?
        - A **type** is just a predicate on the set of values in a system
            - the type `int` can be thought of as a boolean function that returns `true` on integers and `false` otherwise
            - equivalently, we can think of a type as just a **subset of all values**
        - For efficiency and tractability, the predicates are usually very simple
            - types are an **abstraction** mechanism
        - We can easily add new types that distinguish different subsets of values

```
type tp =
    | IntT                  (* type of integers *)
    | PosT | NegT | ZeroT   (* refinements of ints *)
    | BoolT                 (* type of booleans *)
    | TrueT | FalseT        (* subsets of booleans *)
    | AnyT                  (* any value *)
```

    - Modifying the typing rules
        - We need to refine the typing rules
        - Some easy cases (just split up the integers into their refined cases)

P-INT

$$\frac{i > 0}{E \vdash i : \text{Pos}}$$

N-INT

$$\frac{i < 0}{E \vdash i : \text{Neg}}$$

ZERO

$$\frac{}{E \vdash 0 : \text{Zero}}$$

        - Same for booleans

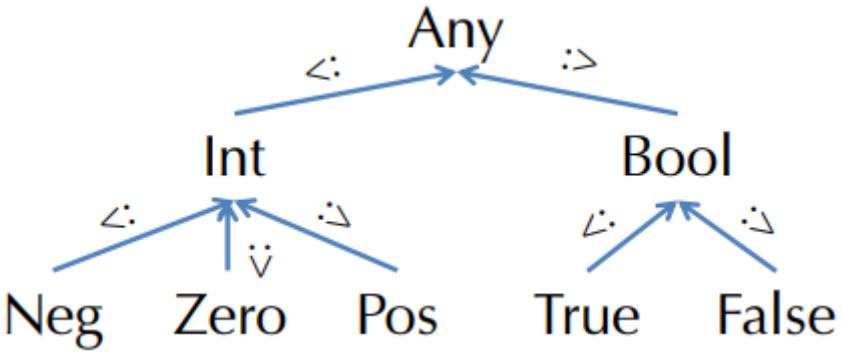
TRUE $\quad\quad$ FALSE

$$\frac{}{E \vdash \text{true} : \text{True}} \qquad \frac{}{E \vdash \text{false} : \text{False}}$$

- What about `if`?
  - Two easy cases

IF-T $\quad \dfrac{E \vdash e_1 : \text{True} \quad E \vdash e_2 : T}{E \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : T}$ $\qquad$ IF-F $\quad \dfrac{E \vdash e_1 : \text{False} \quad E \vdash e_3 : T}{E \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : T}$

  - What if we don't know statically which branch will be taken, that is consider the problem `x:bool` $\vdash$ `if (x) 3 else -1` :?
  - The true branch has type `Pos`, while the false branch has type `Neg`, what should the result type of the whole `if`? `Int` is a safe answer
- Subtyping and Upper Bounds
  - If we view types as sets of values, there is a natural inclusion relation: `Pos` $\subseteq$ `Int`
  - This subset relation gives rise to a **subtype** relation: `Pos` <: `Int`
  - Such inclusions give rise to a **subtyping hierarchy**



  - For types T_1, T_2, define their **least upper bound** w.r.t the hierarchy
- `if` Typing Rule revisited
  - For statically unknown conditionals, we want the return value to be the LUB of the types of the branches

IF-BOOL $\quad \dfrac{E \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_1 \qquad E \vdash e_3 : T_2}{E \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : \mathbf{LUB(T_1, T_2)}}$

    - Note that $\texttt{LUB}(T_1, T_2)$ is the most precise type that is able to describe any value that has either type T_1 or type T_2
    - In math notation, least upper bound is sometimes written $T_1 \vee T_2$
    - LUB is also called the **join** operation
- Subtyping Hierarchy
    - the subtyping relation is a partial order
        - **Reflexive**: $T <: T$ for any type T
        - **Transitive**: $T_1 <: T_2$ and $T_2 <: T_3$ then $T_1 <: T_3$
        - **Antisymmetric**: If $T_1 <: T_2$ and $T_2 <: T_1$ then $T_1 = T_2$
- Soundness of subtyping relations
    - A subtyping relation $T_1 <: T_2$ is **sound** if it approximates the underlying semantic subset relation
    - Formally, we write $[T]$ for the subset of (closed) values of type T
        - $[T] = \{v | \vdash v : T\}$
    - If $T_1 <: T_2$ implies $[T_1] \subseteq [T_2]$, then $T_1 <: T_2$ is sound
- Soundness of LUBs
    - $[T_1] \cup [T_2] \subseteq [\texttt{LUB}(T_1, T_2)]$
    - LUB is an over approximation of the "semantic union"
    - Using LUBs in the typing rules yields **sound approximations** of the program behavior

      It just so happens that LUBs on types <: Int correspond to +

      ADD
      $$\frac{E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \qquad T_1 <: \text{Int} \quad T_2 <: \text{Int}}{E \vdash e_1 + e_2 : T_1 \vee T_2}$$
- Subsumption Rule
    - When we add subtyping judgements of the form $T <: S$ we can uniformly integrate it into the type system generically

      SUBSUMPTION
      $$\frac{E \vdash e : T \quad T <: S}{E \vdash e : S}$$

- Subsumption allows any value of type T to be treated as an S whenever $T <: S$
- Adding this rule makes the search for typing derivations more difficult
    - this rule can be applied anywhere, since $T <: T$
    - Careful engineering of the type system can incorporate the rule into a deterministic algorithm

Nov. 8th - Lecture 16: Subtying; OO: Dynamic Dispatch and Inheritance

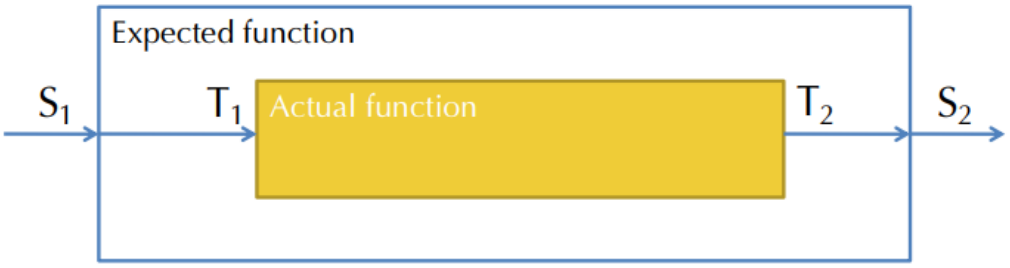
- Recap
    - Downcasting
        - What happens if we have an `Int`, but need something of type `Pos`
            - at compile time, we do not know whether the `Int` is greater than 0
            - at run time, we then do know
        - Add a "**checked downcast**"

$$\frac{E \vdash e_1 : \text{Int} \qquad E, x : \text{Pos} \vdash e_2 : T_2 \qquad E \vdash e_3 : T_3}{E \vdash \text{ifPos } (x = e_1)\ e_2 \text{ else } e_3 : T_2 \vee T_3}$$

        - At runtime, `ifPos` checks whether $e_1 > 0$
            - if yes, branch to e_2; otherwise branch to e_3
        - Inside the expression e_2, x is the name for e_1's value, which is known to be strictly positive because of the dynamic check
        - Note such rules force the programmer to add the appropriate checks
            - we could give integer division the type: `Int` $\rightarrow$ `NonZero` $\rightarrow$ `Int`
- Subtying other types
    - Extending Subtyping to Other Types
        - What about subtyping for tuples?
            - Intuition: whenever a program expects something of type $S_1 * S_2$, it is sound to give it a $T_1 * T_2$

$$\frac{T_1 <: S_1 \quad T_2 <: S_2}{(T_1 * T_2) <: (S_1 * S_2)}$$

- Subtyping for Function Types



  - Need to convert an S_1 to a T_1, and T_2 to S_2, so the argument type is **contravariant** and the output type is **covariant**

$$\frac{S_1 <: T_1 \quad T_2 <: S_2}{(T_1 \text{ -> } T_2) \;<:\; (S_1 \text{ -> } S_2)}$$

- Immutable Records
  - Record type: $\{\texttt{lab}_1 : T_1; \ldots; \texttt{lab}_n : T_n\}$
    - Each $\texttt{lab}_i$ is a label drawn from a set of identifiers

$$\text{RECORD}\quad \frac{E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \quad \ldots \quad E \vdash e_n : T_n}{E \vdash \{lab_1 = e_1;\ lab_2 = e_2;\ \ldots\ ;\ lab_n = e_n\} : \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}}$$

$$\text{PROJECTION}\quad \frac{E \vdash e : \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}}{E \vdash e.lab_i : T_i}$$

- Immutable Record Subtyping
  - Depth subtyping: corresponding fields may be subtypes

$$\text{DEPTH}\quad \frac{T_1 <: U_1 \quad T_2 <: U_2 \quad \ldots \quad T_n <: U_n}{\{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\} <: \{lab_1{:}U_1;\ lab_2{:}U_2;\ \ldots\ ;\ lab_n{:}U_n\}}$$

  - Width subtyping: subtype record may have **more fields**

WIDTH $m \leq n$

$\{lab_1:T_1; lab_2:T_2; \ldots ; lab_n:T_n\} <: \{lab_1:T_1; lab_2:T_2; \ldots ; lab_m:T_m\}$

- Depth & Width Subtyping vs. Layout
  - Width subtyping without depth is compatible with "inlined" record representation as with C structs

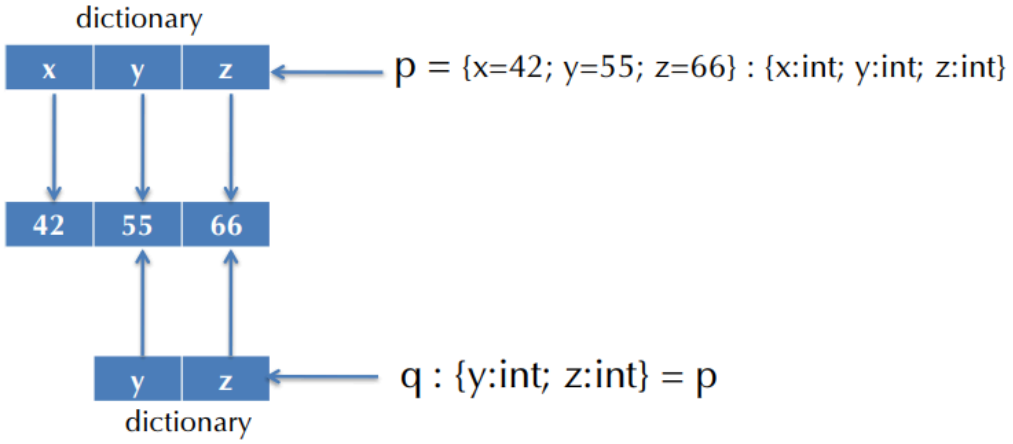
```
{x:int; y:int; z:int}    <:    {x:int; y:int}
```

[Width Subtyping]

    - the layout and underlying fied indices for 'x' and 'y' are identical
    - the 'z' field is just ignored
  - Depth subtyping without width is similarly compatible, assuming that the space used by A is the same as the space used by B whenever $A <: B$
- Immutable Record Subtyping (cont'd)
  - Width subtyping assumes implementations where order of fields matters
    - So, $\{x : \texttt{int}; y : \texttt{int}\} \neq \{y : \texttt{int}; x : \texttt{int}\}$
    - But, $\{x : \texttt{int}; y : \texttt{int}; z : \texttt{int}\} <: \{x : \texttt{int}; y : \texttt{int}\}$
    - Implementation: a record is a struct; subtypes just add fields at the end of the struct
  - Alternative: allow permutation of record fields - $\{x : \texttt{int}; y : \texttt{int}\} = \{y : \texttt{int}; x : \texttt{int}\}$
    - Implementation: compiler sorts the fields before code generation
    - Need to know **all** of the fields to generate the code
  - Permutation is not directly compatible with width subtyping

{x:int; z:int; y:int} = {x:int; y:int; z:int} </: {y:int; z:int}

- If we want both...
  - permutability and dropping, we need to
    - either copy (to rearrange the fields)
    - or use a dictionary



- Mutability and Subtyping
  - Null
    - What is the type of `null`

```
int[] a  = null;  // OK
int x    = null;  // not OK
string s = null;  // OK
```

$$\frac{}{E \vdash \text{null} : r}\ \text{NULL}$$

    - Null has any **reference type**, is **generic**
    - Type safety?
      - requires defined behavior when dereferencing `null`
      - requires a safety check for every dereference operation (typically implemented using low-level hardware "trap" mechanisms)
  - Subtyping and References
    - Proper subtyping relationship for **references** and **arrays**?
    - Suppose we have the type for division operation as `Int` → `NonZero` → `Int`
    - Should (`NonZero ref`) <: (`Int ref`)
    - But consider

```
Int bad(NonZero ref r) {
  Int ref a = r;     (* OK because NonZero ref <: Int ref *)
  a := 0;            (* OK because 0 : Zero <: Int *)
  return (42 / !r)   (* OK because !r has type NonZero *)
}
```

  - Mutable Structures are Invariant
    - Covariant reference types are unsound, as shown previously

- Contravariant reference types are also unsound (if $T_1 <: T_2$, then `ref` $T_2 <:$ `ref` T_1 is also unsound)

```
Assume: NonZero <: Int    =>    ref Int <: ref NonZero

Int ref a;
a := 0;
NonZero ref b;
b = a;
return (1 / !b);
```

- Moral: T_1 `ref` $<: T_2$ `ref` implies $T_1 = T_2$
- Same holds for arrays, OCaml-style mutable records, object fields, etc.
    - Note: Java and C# get this wrong
    - They allow covariant array subtyping
    - But compensate by adding a dynamic check on **every** array update

```
Assume: B <: A => B[] <: A[]

B[] b = new B[5];
A[] a = b;
a[0] = new A;
b[0].something in b
```

- Another Way to See it
    - We can think of a reference cell as an immutable record (object) with 2 functions and some hidden state

      T `ref` $\simeq \{$`get:` `unit` $\to T;$ `set:` $T \to$ `unit`$\}$

        - `get` returns the value hidden in the state
        - `set` updates the value hidden in the state
    - When is T `ref` $<: S$ `ref`
    - Records, like tuples, subtyping extends pointwise over each component
    - {get: unit -> T; set: T -> unit} <: {get: unit -> S; set: S -> unit}
        - get components are subtypes: unit -> T <: unit -> S
          set components are subtypes: T -> unit <: S -> unit
    - From `get`, we must have $T <: S$; `set`, we must have $S <: T$
    - We conclude $T = S$

- Structural vs. Nominal Types
    - Structural vs. Nominal typing
        - Type equality/subsumption defined by the **structure** or **name** of the data?
        - ```
          (* OCaml: *)
          type cents = int    (* cents = int in this scope *)
          type age = int

          let foo (x:cents) (y:age) = x + y
          ```

          ```
          (* Haskell: *)
          newtype Cents = Cents Integer  (* Integer and Cents are
                                            isomorphic, not identical *)
          newtype Age = Age Integer

          foo :: Cents -> Age -> Int
          foo x y = x + y                (* Ill typed! *)
          ```
        - Type abbreviations are treated "structurally", Newtypes are treated by "name"
    - Nominal Subtyping in Java
        - In Java, classes and interfaces must be named; their relationships are explicitly declared

          ```
          (* Java: *)
          interface Foo {
            int foo();
          }

          class C {      /* Does not implement the Foo interface */
            int foo() {return 0;}
          }

          class D implements Foo {
            int foo() {return 1;}
          }
          ```
        - Similarly for inheritance: programmers must declare the subclass relation via the "`extends`" keyword; type-checker still checks that the classes are structurally compatible
- OAT's Type System
    - OAT's Treatment of Types
        - Primitive (non-reference) types: `int, bool`
        - Definitely non-null reference types: `R`
            - named mutable structs with **width subtyping**
            - strings
            - arrays (including length information)
        - Possibly-null reference types: `R?`
            - subtyping: `R` <: `R?`
          ```

- checked downcast syntax `if?`

```
int sum(int[]? arr) {
  var z = 0;
  if?(int[] a = arr) {
    for(var i = 0; i<length(a); i = i + 1;) {
      z = z + a[i];
    }
  }
  return z;
}
```

- OAT features

  Named structure types with mutable fields
  - but using structural, width subtyping

  Typed function pointers

  Polymorphic operations: length, and == or !=
  - need special case handling in the type-checker

  Type-annotated null values: **t null** always has type **t?**

  Definitely-not-null values => "atomic" array initialization syntax
  As an example
  - null is not allowed as a value of type int[]
  - So to construct a record containing a field of type int[], need to initialize it

- OAT "Returns" Analysis
  - typesafe, statement-oriented imperative languages must ensure a function (always) returns a value of the appropriate type
    - does the returned expression's type match the one declared by the function?
    - do all paths through the code return appropriately?
  - OAT's statement checking judgement
    - takes the expected return type as input
    - produces a boolean flag as output

- Compiling classes and objects
  - Code generation for objects
    - Classes
      - Generate data structure types (for objects that are instances of the class and for the class tables)
      - Generate the class tables for dynamic dispatch
    - Methods

        - method body code is similar to functions/closures
        - method calls require **dispatch**
    - Fields
        - Issues are the same as for records
        - generating access code
    - Constructors
        - object initialization
    - Dynamic types
        - checked downcasts
        - "instanceof" and similar type dispatch
- Multiple Implementations
    - The same interface can be implemented by multiple classes

```
interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
    public int size();
}
```

```
class IntSet1 implements IntSet {
  private List<Integer> rep;
  public IntSet1() {
    rep = new LinkedList<Integer>();}

  public IntSet1 insert(int i) {
     rep.add(new Integer(i));
    return this;}

  public boolean has(int i) {
    return rep.contains(new Integer(i));}

  public int size() {return rep.size();}
}
```

```
class IntSet2 implements IntSet {
  private Tree rep;
  private int size;
  public IntSet2() {
    rep = new Leaf(); size = 0;}

  public IntSet2 insert(int i) {
    Tree nrep = rep.insert(i);
    if (nrep != rep) {
      rep = nrep; size += 1;
    }
    return this;}

  public boolean has(int i) {
    return rep.find(i);}

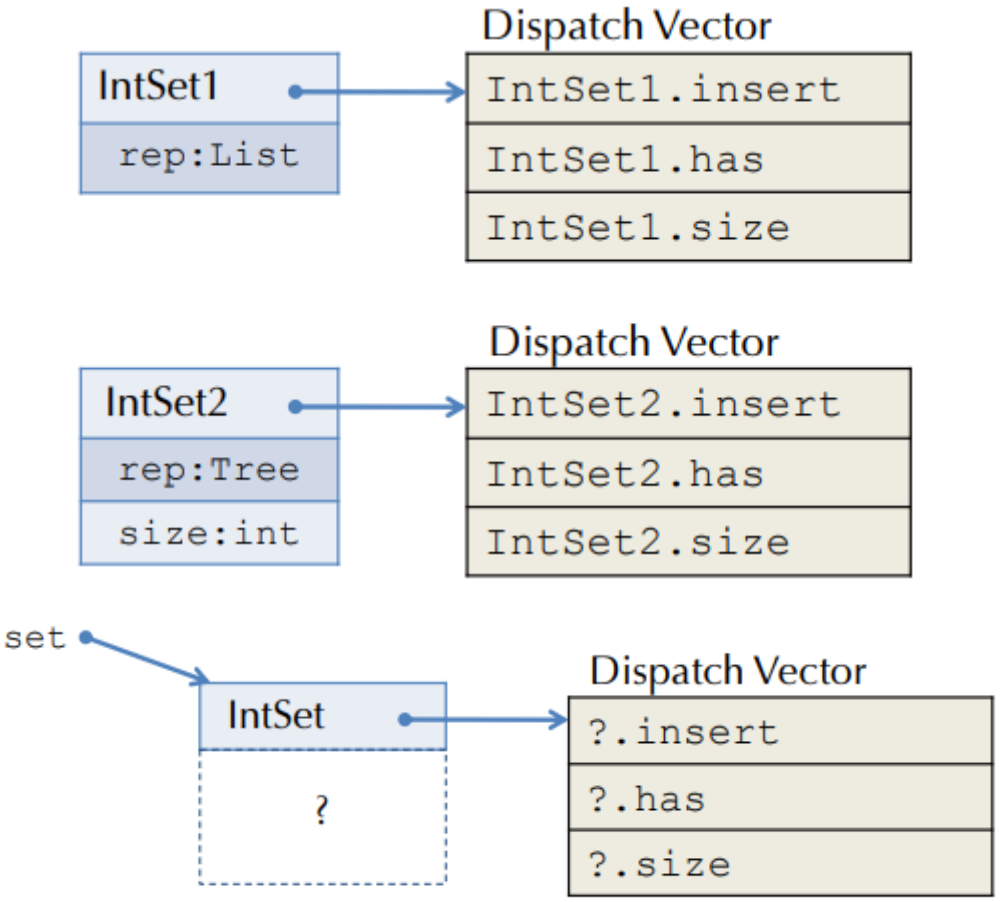
  public int size() {return size;}
}
```


- The Dispatch problem
    - Consider a client program with the above interface

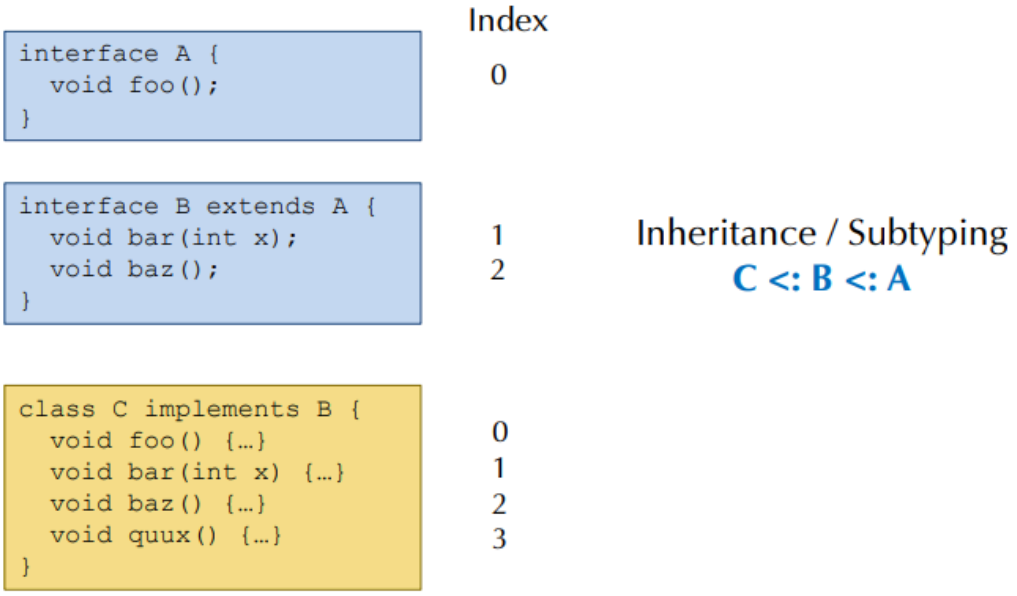
```
IntSet set = ...;
int x = set.size();
```

    - Which code to call?
    - Client code does not know the answer, so objects must "know" which code to call, invocation of a method must indirect through the object
- Compiling objects

- Objects contain a pointer to a **dispatch vector** (also called a **virtual table** or **vtable**) with pointers to method code
- Code receiving `set:IntSet` only knows that `set` has an initial dispatch vector pointer and the layout of that vector

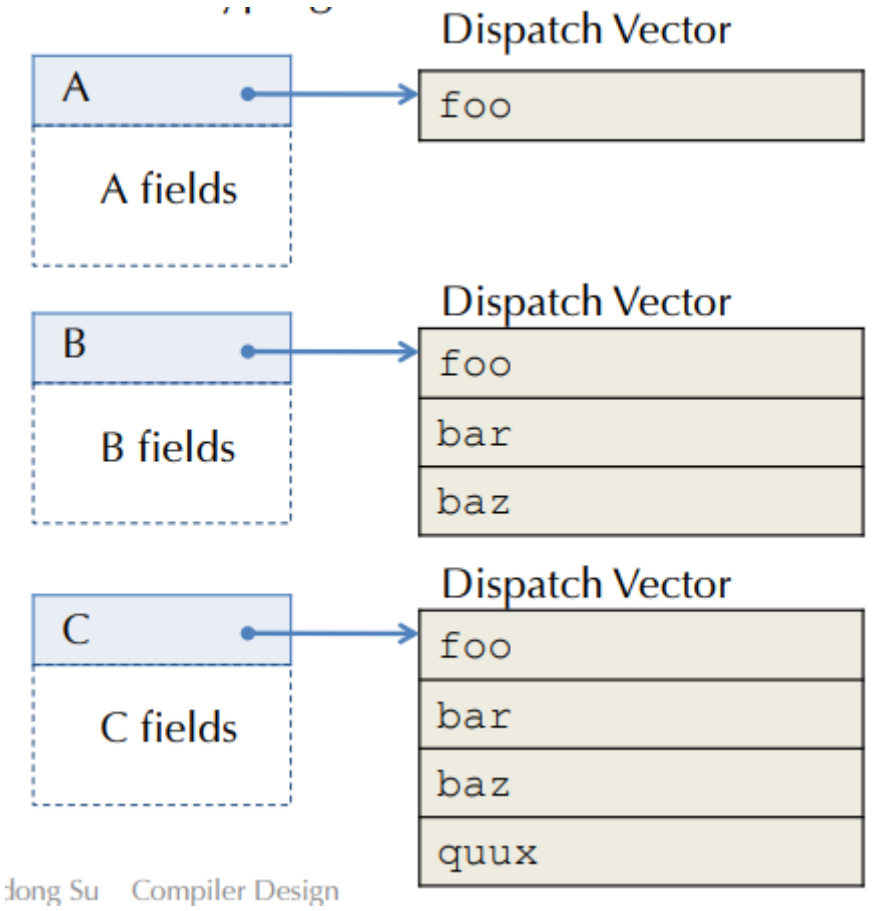


- Method Dispatch (Single Inheritance)
  - Idea: every method has its own small integer index
  - Index is used to look up the method in the dispatch vector



- Dispatch Vector Layouts
  - Each interface and class gives rise to a dispatch vector layout

- Note that inherited methods have identical dispatch indices in the subclass (width subtyping)



Nov. 13th - Lecture 17: Multiple Inheritance & Optimization I

- Recap
  - Static vs. Dynamic Dispatch
    - Consider

```
A* p = new B;
p -> foo();
```

    - Static dispatch: use `p`'s static type `A`'s `foo()`

      Dynamic dispatch: use `p`'s dynamic type `B`'s `foo()`
- Multiple Inheritance
  - C++
    - A class may declare more than one superclass
    - Semantic problem: ambiguity

```
class A { int m(); }
class B { int m(); }
class C extends A, B {...}
```

- Same problem can happen with fields
- In C++, fields/methods can be duplicated when such ambiguities arise (explicit sharing can also be declared)

- Java
  - a class may implement more than one interface
  - no semantic ambiguity when two interfaces declare the same method (the class will implement a single method)

```
interface A { int m(); }
interface B { int m(); }
class C implements A, B { int m() {...}}
```

- Dispatch Vector Layout Strategy Breaks

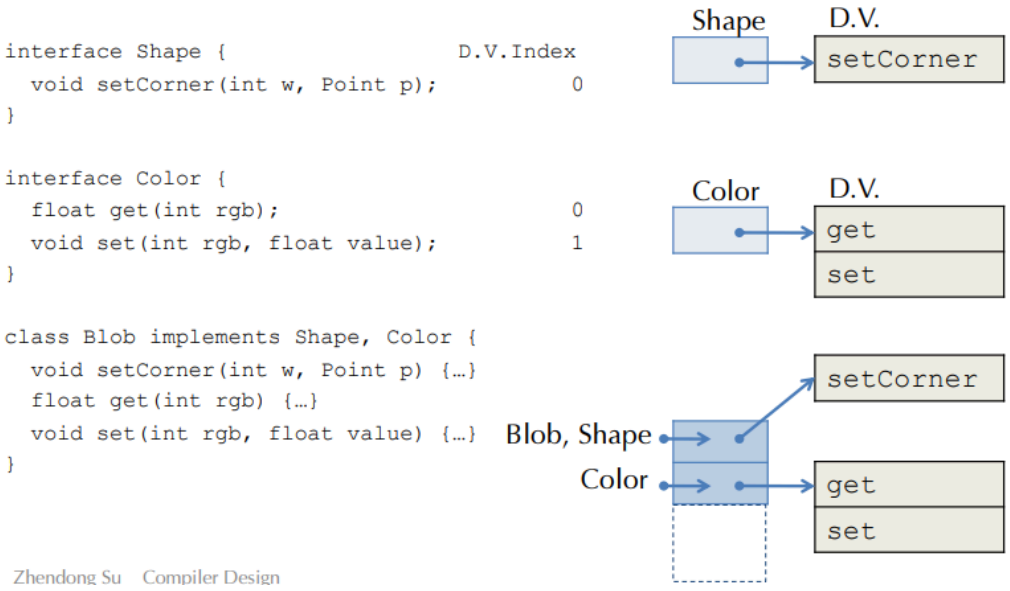
```
interface Shape {                          D.V.Index
  void setCorner(int w, Point p);              0
}

interface Color {
  float get(int rgb);                          0
  void set(int rgb, float value);              1
}

class Blob implements Shape, Color {
  void setCorner(int w, Point p) {…}           0?
  float get(int rgb) {…}                       0?
  void set(int rgb, float value) {…}           1?
}
```

- General Approaches
  - Problem: Cannot directly identify methods by position anymore
  - Option 1: Allow multiple dispatch vector tables
    - Choose which D.V. to use based on static type
    - Casting from/to a class may require runtime operations
  - Option 2: Use a level of indirection

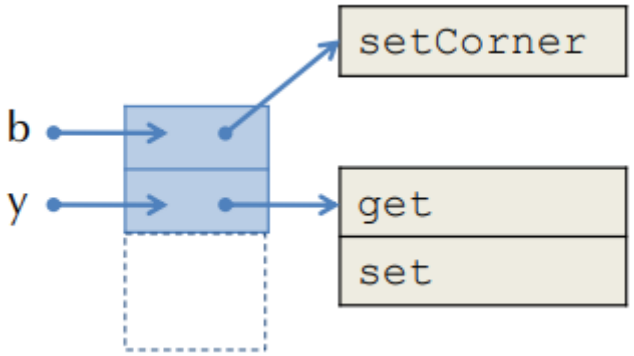
        - Map method identifiers to code pointers (e.g. indexed by method name)
        - Use a hash table
        - May need to search up the class hierarchy
    - Option 3: Give up separate compilation
        - Use "sparse" dispatch vectors, or binary decision trees
        - Must know the entire class hierarchy
    - Different Java compilers pick different approaches to options 2 & 3
- Option 1: Multiple Dispatch Vectors
    - Duplicate the D.V. pointers in the object representation
    - Static type of the object determines which D.V. is used



- Multiple D.V.
    - An object may have multiple "entry points"
        - Each entry point corresponds to a dispatch vector
        - Which one to use depends on the object's static type

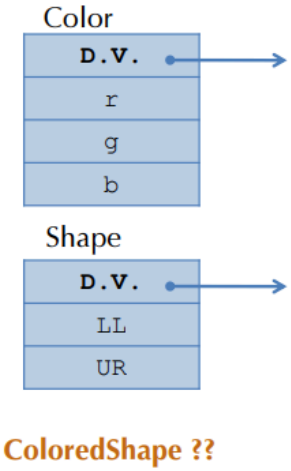
```
Blob b = new Blob();
Color y = b;        // implicit cast
```

    - Compile `Color y = b;` → `Movq [b]+8, y`



- Multiple D.V. Summary
    - Pros: efficient dispatch; similar cost as for single inheritance
    - Cons: cast has a runtime cost; more complicated programming model; hard to understand/debug
- Multiple Inheritance: Fields
    - Multiple supertypes (Java): methods conflict
    - Multiple inheritance (C++): fields can also conflict
        - fields can no longer be constant offsets from the start of the object

```
class Color {
  float r, g, b; /* offsets: 4,8,12 */
}
class Shape {
  Point LL, UR; /* offsets: 4, 8 */
}
class ColoredShape :
  public Color, public Shape {
  int z;
}
```



- vtable for C++ Multiple Inheritance

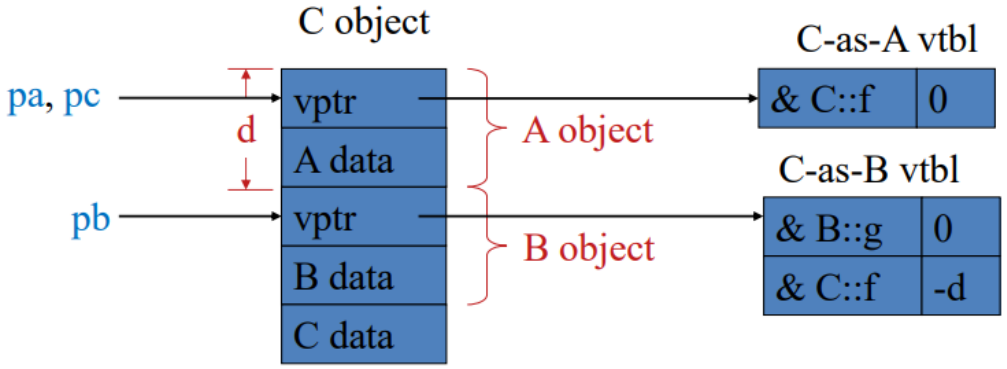
```
class A {
  public:
    int x;
    virtual void f();
};
class B {
  public:
    int y;
    virtual void g();
    virtual void f();
};
```

```
class C: public A, public B {
  public:
    int z;
    virtual void f();
};

C *pc = new C;
B *pb = pc;
A *pa = pc;
```

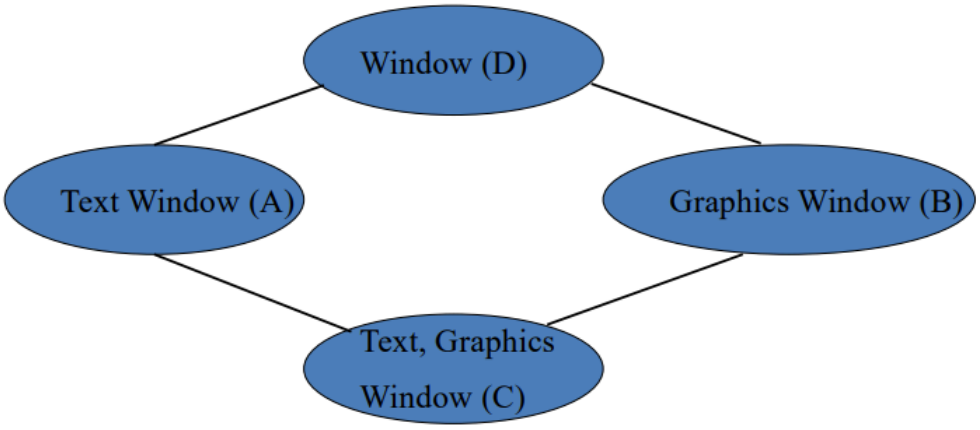
Three pointers to the same object, but different static types

- Objects & Classes

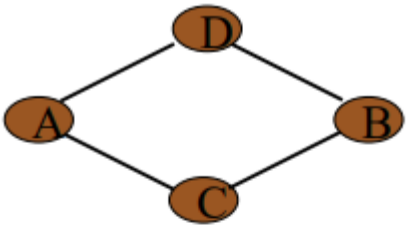


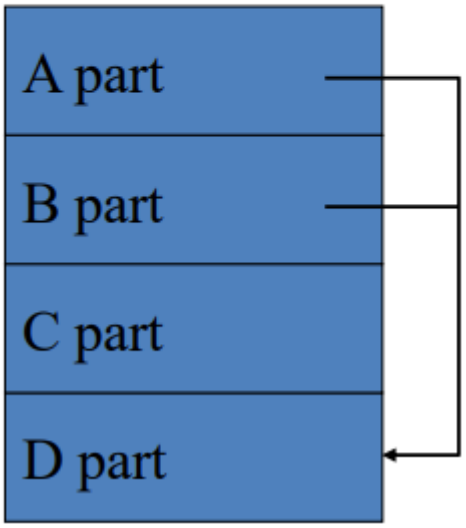
  - Call to `pc -> g` can proceed through C-as-B vtbl
  - Offset *d* in vtbl is used in call to `pb -> f`
    - Since `C::f` may refer to A data that is above the pointer `pb`

- Multiple Inheritance "Diamond"

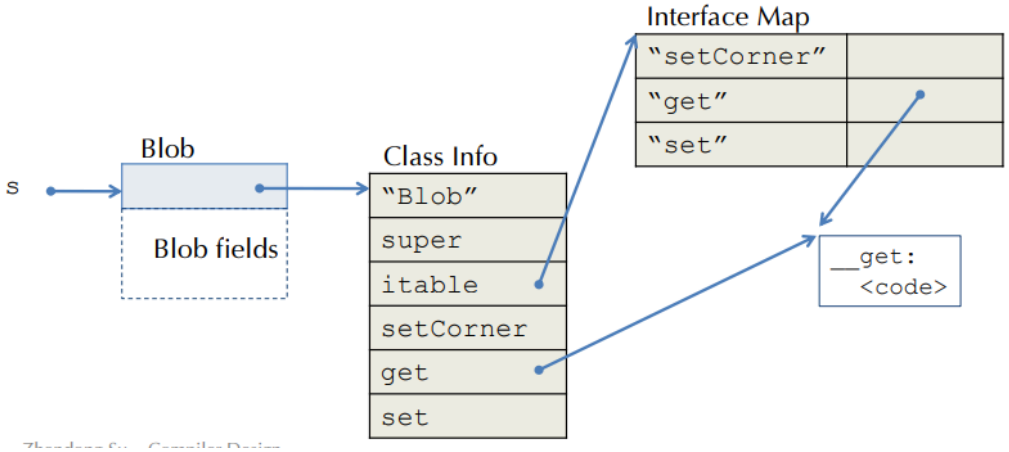


- Diamond Inheritance in C++
  - Standard base classes
    - D members appear twice in C
  - Virtual base classes
    - `class A: public virtual D {...}`
    - Avoid duplication of base class members
    - Require additional pointers so that D part of A, B parts of object can be shared
  - Multiple Inheritance is complicated in C++
    - Because of **its desire for efficient lookup**





- Option 2: Search + Inline Cache
    - For each class/interface, keep a table:
      `method names` → `method code`
        - recursively walk up the hierarchy looking for the method name
    - Note
        - Identifiers in quotes are not strings
        - In practice, they are some kind of unique identifiers



- Why is search necessary?

```
interface Incrementable {
    public void inc();
}
class IntCounter implements Incrementable {
    public void add(int);
    public void inc();
    public int value();
}
class FloatCounter implements Incrementable {
    public void inc();
    public void add(float);
    public float value();
}
```

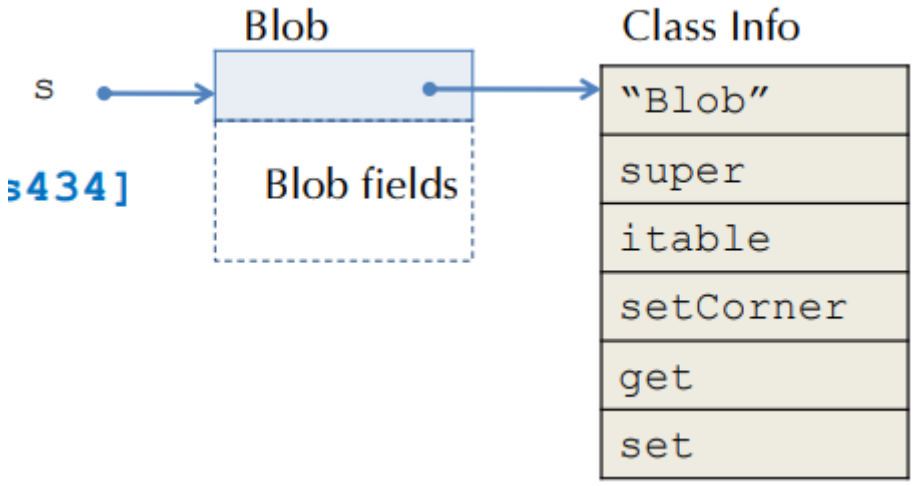
void add2(Incrementable x) { x.inc(); x.inc(); }

- Inline Cache code
    - Optimization
        - at call site, store class and code pointer in a cache
        - on method call, check whether class matches cached value
    - Compiling: `Shape s = new Blob(); s.get();`
    - Compiler knows that `s` is a `Shape`
        - Suppose `%rax` holds object pointer
    - Cached interface dispatch

```
// set up parameters
                                    s
 movq [%rax], tmp
 cmpq tmp, [cacheClass434]
 Jnz __miss434
 callq [cacheCode434]
__miss434:
 // do the slow search
```

Table in data seg.

```
cacheClass434:
  "Blob"
cacheCode434:
  <ptr>
```



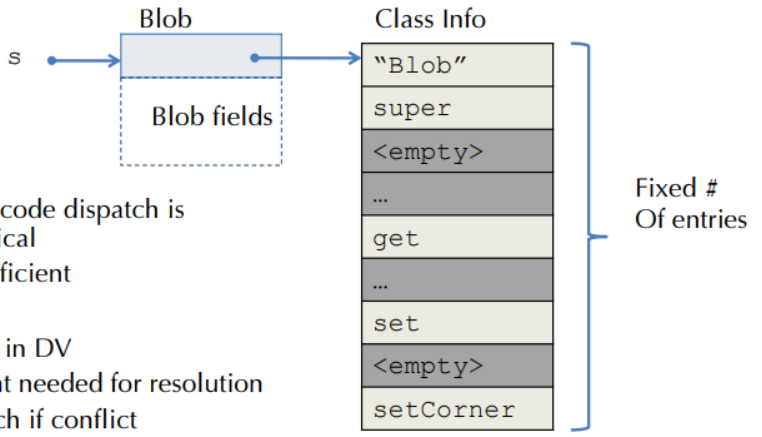
- Option 2 variant 2: Hash table
    - Idea: don't try to give all methods unique indices
        - resolve conflicts by checking that the entry is correct at dispatch
    - Using hashing to generate indices
        - range of the hash values should be relatively small
        - hash indices can be pre computed, but passed as an extra parameter

```
interface Shape {                       D.V.Index
  void setCorner(int w, Point p);       hash("setCorner") = 11
}

interface Color {
  float get(int rgb);                   hash("get") = 4
  void set(int rgb, float value);       hash("set") = 7
}

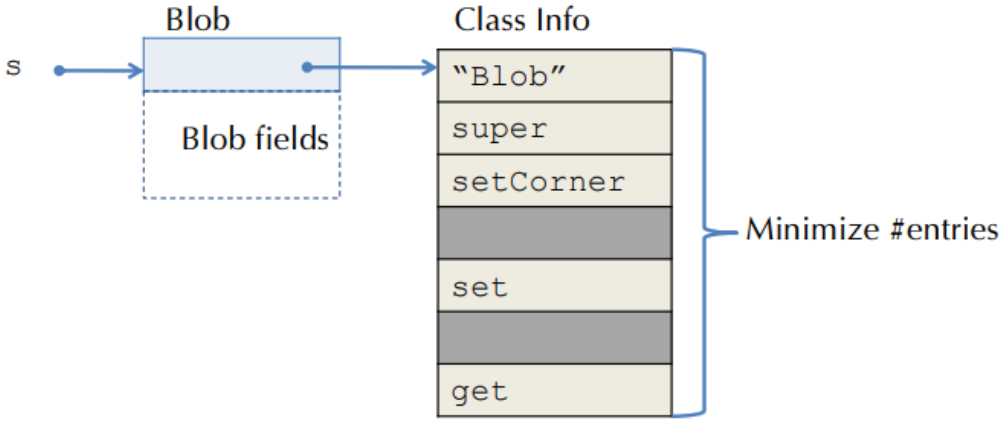
class Blob implements Shape, Color {
  void setCorner(int w, Point p) {…}        11
  float get(int rgb) {…}                    4
  void set(int rgb, float value) {…}        7
}
```

- Dispatch with Hash Tables
    - What if there is a conflict?
        - entries containing several methods point to code that resolves conflict (e.g. by searching through a table based on class name)



- Advantage
  - Simple, basic code dispatch is (almost) identical
  - Reasonably efficient
- Disadvantage
  - Wasted space in DV
  - Extra argument needed for resolution
  - Slower dispatch if conflict

- Option 3 variant 1: Sparse D.V. Tables
  - Give up on separate compilation, we have access to whole class hierarchy
  - Ensure no 2 methods in same class are allocated the same D.V. offset
    - allow holes in the D.V. just like the hash table solution
    - Unlike hash table, there is never a conflict
  - Compiler needs to construct the method indices
    - graph coloring can be used to construct D.V. layouts reasonably efficiently (to minimize size)
    - Finding an optimal solution is NP complete
- Example Object Layout
  - Advantage: identical dispatch & performance to single-inheritance case
  - Disadvantage: must know entire class hierarchy

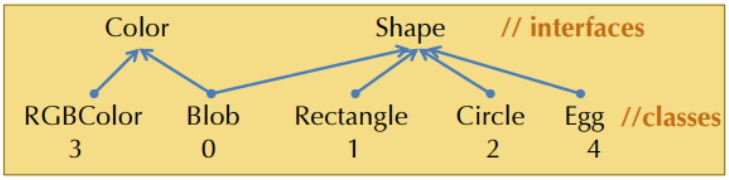


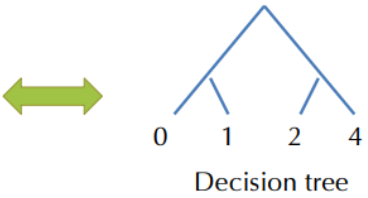
- Option 3 variant 2: BST

- **Idea**: Use conditional branches (not indirect jumps)
- Each object has a class index (unique per class) as first word
  - Instead of D.V. pointer (no need for one!)
- Method invocation uses range tests to select among **n** possible classes in **log n** time
  - Direct branches to code at the leaves

```
Shape x;
x.SetCorner(…);
```

```
Mov eax, [x]
Mov ebx, [eax]
Cmp ebx, 1
Jle  __L1
Cmp ebx, 2
Je __CircleSetCorner
Jmp __EggSetCorner
__L1:
Cmp ebx, 0
Je __BlobSetCorner
Jmp __RectangleSetCorner
```



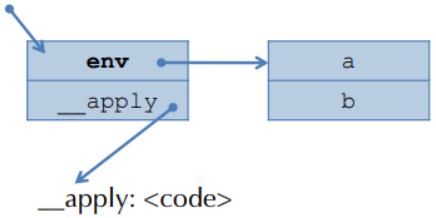


Decision tree

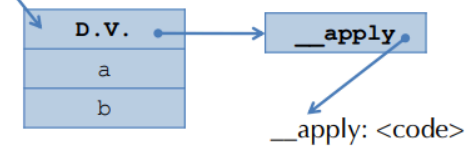
- Search Tree Tradeoffs
  - Binary decision trees work well if the distribution of classes that may appear at a call site is skewed
    - Branch prediction hardware eliminates branch stall of ~10 cycles (on x86)
  - Profiling helps find the common paths for each call site individually
    - Put the common case at the top of the decision tree (so less search)
    - 90/10 rule of thumb: 90% invocations at a call site go to the same class
  - Drawbacks
    - Like sparse D.V.'s, one needs the entire class hierarchy
      - To know how many leaves are needed in the search tree
    - Indirect jumps can have better performance if there are > 2 classes
      - At most one misprediction
- Observe: Closure ≈ single-method Object

Free variables	≈	Fields
Environment pointer	≈	"this" parameter
Closure for function	≈	Instance of this class

```
fun (x,y) -> x + y + a + b
```




```
class C {
  int a, b;
  int apply(x,y) {
    x + y + a + b
  }
}
```



- Classes and Objects in LLVM
  - Representing Classes in the LLVM
    - During type-checking, create a class hierarchy

        - maps each class to its interface
        - superclass, constructor type, fields, method types (plus whether they inherit & which class they inherit from)
    - Compile the class hierarchy to produce
        - an LLVM IR struct type for each object instance
        - an LLVM IR struct type for each vtable (a.k.a class table)
        - global definitions that implement the class tables
- Example OO Code (Java)

```
class A {
  A (int x)                 // constructor
  { super(); int x = x; }

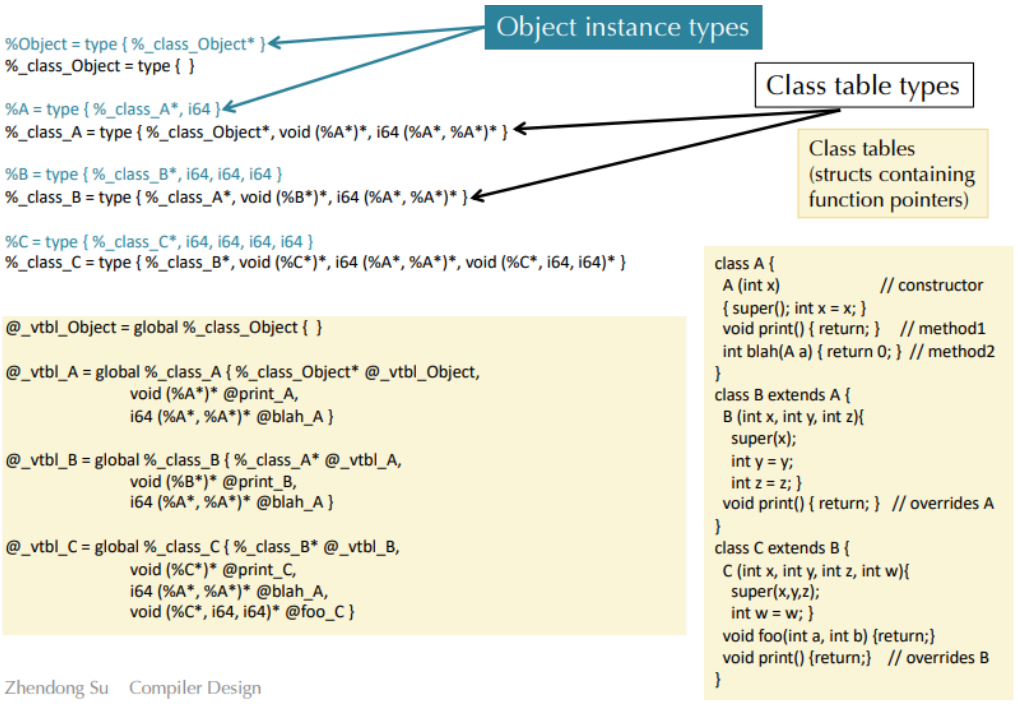
  void print() { return; }    // method1
  int blah(A a) { return 0; } // method2
}

class B extends A {
  B (int x, int y, int z){
    super(x);
    int y = y;
    int z = z;
  }

  void print() { return; }  // overrides A
}

class C extends B {
  C (int x, int y, int z, int w){
    super(x,y,z);
    int w = w;
  }
  void foo(int a, int b) {return;}
  void print() {return;}   // overrides B
}
```

- Example OO hierarchy in LLVM



- Method arguments
  - method bodies are compiled just like top-level procedures
  - except that they have an implicit extra argument: **`this`** or **`self`**
    - Historically, these were called the "receiver object"
    - method calls were thought of sending "messages" to "receivers"

A method in a class

```
class IntSet1 implements IntSet {
   …
  IntSet1 insert(int i) { <body> }
}
```

… is compiled like this (top-level) procedure

```
IntSet1 insert(IntSet1 this, int i) { <body> }
```

  - Note
    - The type of "**`this`**" is the class containing the method
    - References to fields inside <body> are compiled like **`this.field`**

- LLVM Method Invocation Compilation

Consider method invocation $[\![H;G;L \vdash e.m(e_1,\ldots,e_n) : t]\!]$

1. Compile $[\![H;G;L \vdash e : C]\!]$
   - To get a (pointer to) an object value of class type C
   - Call this value obj_ptr
2. Use Getelementptr to **extract the vtable pointer** from obj_ptr
3. **Load the vtable pointer**
4. Use Getelementptr to **extract the function pointer's address** from vtable
   - Use the information about C in H
5. **Load the function pointer**
6. Call through the function pointer, passing 'obj_ptr' for this

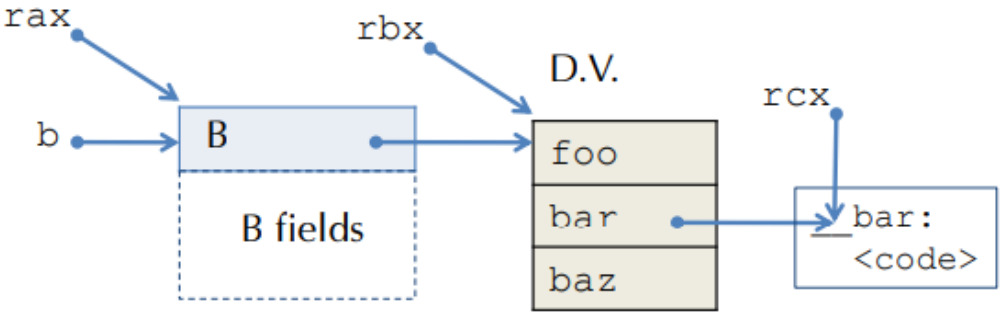
$$\texttt{call (cmp\_typ t) m(obj\_ptr, } [\![e_1]\!], \ldots, [\![e_n]\!])$$

In general, function calls may require **bitcast** to account for subtyping

- "Actual" argument type may be a subtype of the expected "formal" type

Nov. 15th - Lecture 18: Optimization and Data Flow Analysis

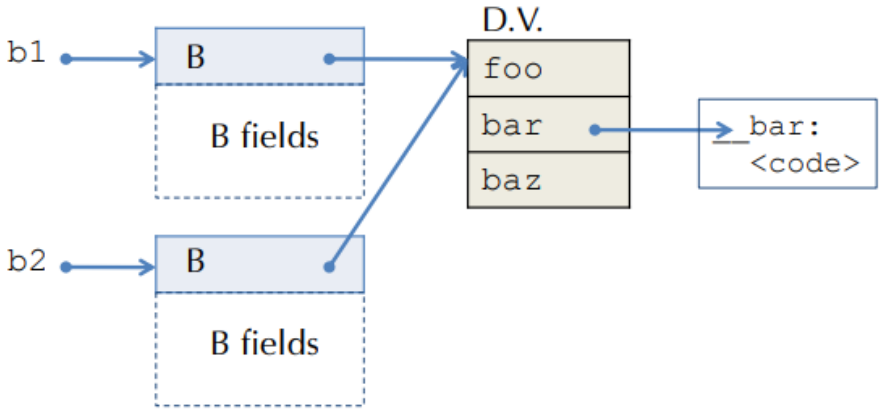
- Recap
  - x86 Code for Dynamic Dispatch
    - Suppose `b: B`
    - What code to generate for `b.bar(3)`
      - `bar` has index 1
      - Offset $= 8 * 1$



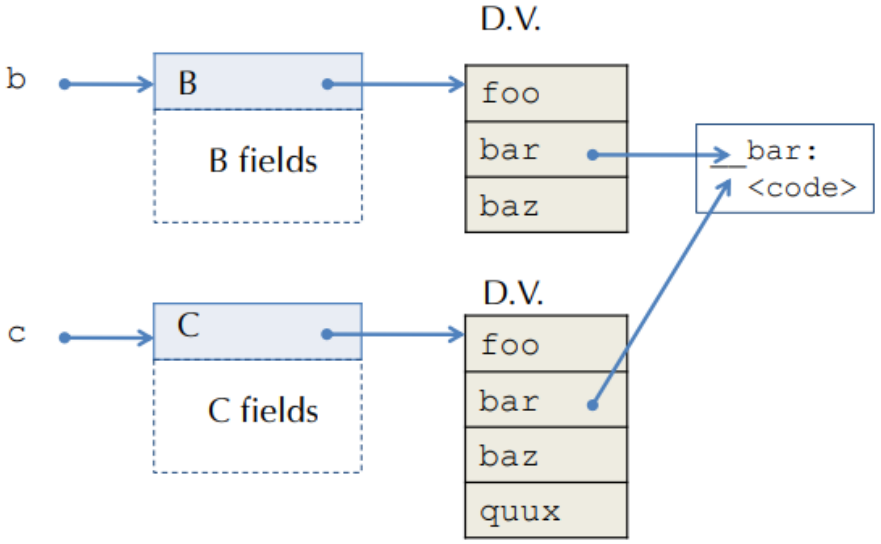

```
movq [[b]], %rax
movq [%rax], %rbx
movq [rbx+8], %rcx      // D.V. + offset
movq %rax, %rdi         // "this" pointer
movq 3, %rsi            // Method argument
call %rcx               // Indirect call
```

  - Sharing Dispatch Vectors
    - All instances of a class may share the same dispatch vector

        - assuming that methods are immutable
    - Code pointers stored in the dispatch vector are available at link time
        - dispatch vectors can be built once at link time



    - One job of object constructor is to fill in the object's pointer to the appropriate dispatch vector
    - D.V.'s address is the runtime representation of the object's type
- Inheritance: Sharing Code
    - Inheritance: Method code "copied down" from the superclass
        - if not overridden in the subclass
    - Works with separate compilation: superclass code not needed



- Compiling Static Methods
    - Java supports **static** methods
        - methods that belong to a class, not the instances of the class
        - They have no "`this`" parameter
    - Compiled exactly like normal top-level procedures

        - No slots needed in the dispatch vectors
        - No implicit "`this`" parameter
    - They are not really methods
        - They can only access static fields of the class
- Compiling Constructors
    - Java and C++ classes can declare constructors that create new objects
        - initialization code may have parameters supplied to the constructor
        - `new Color(r,g,b)`
    - Modula-3: object constructors take no parameters
        - Initialization would typically be done in a separate method
    - Constructors are compiled just like static methods, except
        - The "`this`" variable is initialized to a newly allocated block of memory big enough to hold D.V. pointer + fields according to object layout
        - Constructor code initializes the fields (can use any methods)
        - The D.V. pointer is initialized (before initialization code so that they can be used)
- Compiling Checked Casts
    - How do we compile downcast in general?
      Consider this generalization of Oat's checked cast

      ```
      if? (t x = exp) { … } else { … }
      ```

    - Reason by cases
        - t must be either null, ref or ref? (can't be just int or bool)
    - **If t is null**
        - The static type of exp must be ref? for some ref.
        - If exp == null then take the true branch, otherwise take the false branch
    - **If t is string or t[]**
        - The static type of exp must be the corresponding string? Or t[]?
        - If exp == null take the false branch, otherwise take the true branch
    - **If t is C**
        - The static type of exp must be D or D? (where C <: D)
        - If exp == null take the false branch, otherwise
        - emit code to **walk up the class hierarchy** starting at T to look for C (T is exp's dynamic type)
        - If found, then take true branch else take false branch
    - **If t is C?**
        - The static type of exp must be D? (where C <: D)
        - If exp == null take the true branch, otherwise
        - Emit code to **walk up the class hierarchy** starting at T to look for C (T is exp's dynamic type)
        - If found, then take true branch else take false branch
- "Walking up the Class Hierarchy"

- A non-null object pointer refers to an LLVM struct with a type like

```
%B = type { %_class_B*, i64, i64, i64 }
```

- The first entry of the struct is a pointer to the vtable for Class B
  - This pointer is the **dynamic type** of the object
  - It will have the value @vtbl_B
- The first entry of the class table for B is a pointer to its superclass

```
@_vtbl_B = global %_class_B { %_class_A* @_vtbl_A,
                              void (%B*)* @print_B,
                              i64 (%A*, %A*)* @blah_A }
```

- Therefore, to find out whether an unknown type X is a subtype of C
  - **Assume C is not Object** (ruled out by "silliness" checks for downcast)

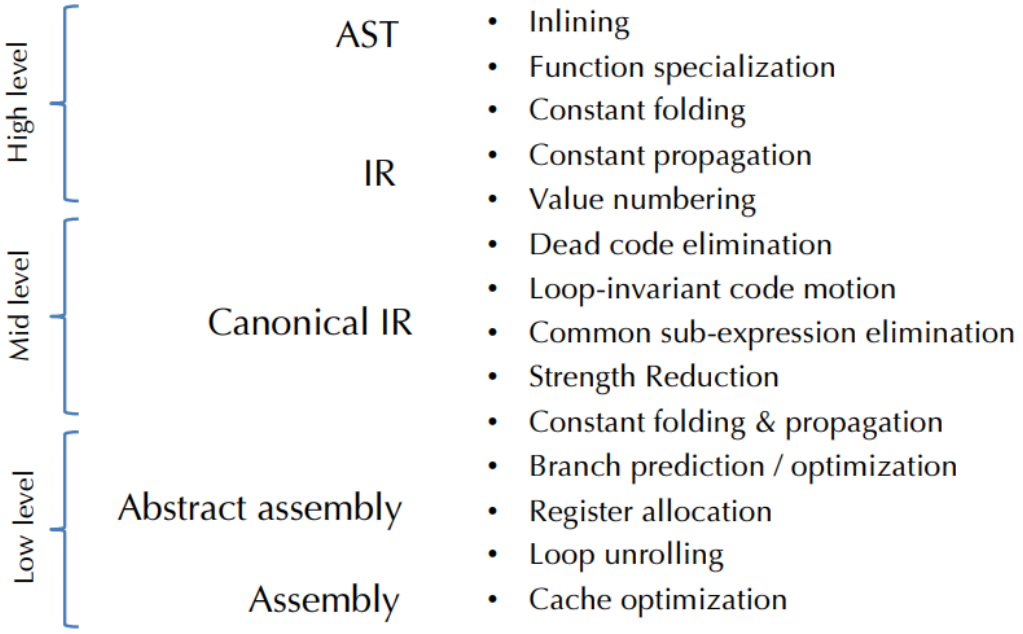
  LOOP
  - If **X == @_vtbl_Object** then NO, X is not a subtype of C
  - If **X == @_vtbl_C** then YES, X is a subtype of C
  - If **X = @_vtbl_D** set X to @_vtbl_E where E is D's parent & goto LOOP

- Optimizations
  - Optimizations
    - The code generated by OAT compiler is inefficient
      - redundant moves + unnecessary arithmetic instructions
    - Consider

```
int foo(int w) {
    var x = 3 + 5;
    var y = x * w;
    var z = y - 0;
    return z * 4;
}
```

  - Why do we need optimizations?
    - To help programmers
      - programmers write modular, clean, high-level programs
      - Compiler generates efficient, high-performance assembly
    - Programmers do not write optimal code
    - High-level PLs make avoiding redundant computation hard
      - `A[i][j] = A[i][j] + 1`
    - Architectural independence

        - optimal code depends on features not expressed to the programmer
        - modern architectures assume optimization
    - Different kinds of optimizations
        - Time: improve execution speed
        - Space: reduce amount of memory needed
        - Power: lower power consumption
- Some caveats
    - Optimizations are code transformations
        - they can be applied at any stage of the compiler
        - they must be safe: should not change the meaning of the program
    - In general, optimizations require some program analysis
        - to determine if the transformation really is safe
        - to determine whether the transformation is cost effective
- When to apply optimization



- Where to optimize
    - Usual goal: improve time performance
    - Problem: many optimizations trade space for time
    - e.g. Loop unrolling

```
for (int i = 0; i < 100; i += 1) {
    s += a[i];
}

for (int i = 0; i < 99; i += 2) {
    s += a[i];
    s += a[i+1];
}
```

- Tradeoffs
    - increasing code space slows down whole program a tiny bit (extra instructions to manage), but speeds up the loop a lot
    - For frequently executed code with long loops, generally a win
    - Interacts with instruction cache and branch prediction hardware
- Complex optimizations may never pay off

- Writing Fast Programs in Practice
    - Pick the right algorithms and data structures
        - these have much bigger impact on performance than optimizations
        - reduce number of operations & memory access
        - Minimize indirection - it breaks working-set coherence
    - Then turn on compiler optimizations
    - Profile to determine program hot spots
    - Evaluate whether the algorithm/data structure design works
    - if so: "tweak" the source code until the optimizer does "the right thing" to the machine code

- Safety
    - Whether an optimization is safe depends on the language semantics
        - languages with weaker guarantees permit more optimizations, but have more ambiguity in their behavior
        - In Java, tail-call optimization is not (yet) valid/supported
        - In C, loading from uninitialized memory is undefined

- e.g. loop-invariant code motion (LICM)
    - idea: hoist invariant code out of a loop

```
while (b) {
    z = y/x;
    ...          // x, y not updated
}

z = y/x;
while (b) {
    ...          // x, y not updated
}
```

    - Not always efficient, not always safe (if `b` is false and `x` is 0, then moving it out does not guarantee safety)
- Constant Folding
    - Idea: if operands are statically known, compute value at compile-time

      `int x = (2 + 3) * y` → `int x = 5 * y`

      `b & false` → `false`
    - Performed at every stage of optimization
        - constant expressions can be created by translation or earlier optimizations
    - e.g. `A[2]` may be compiled to

      `mem[mem[A] + 2 * 4]` → `mem[mem[A] + 8]`
- Constant Folding Conditionals

```
if (true) S            ➔ S
if (false) S           ➔ ;
if (true) S else S'    ➔ S
if (false) S else S'   ➔ S'
while (false) S        ➔ ;

if (2 > 3) S           ➔ ;
```

- Algebraic Simplification

  Identities
  - `a * 1 → a` `a * 0 → 0`
  - `a + 0 → a` `a - 0 → a`
  - `b | false → b` `b & true → b`

  Reassociation & commutativity
  - `(a + 1) + 2 → a + (1 + 2) → a + 3`
  - `(2 + a) + 4 → (a + 2) + 4 → a + (2 + 4) → a + 6`

  Strength reduction: (replace expensive op with cheaper op)
  - `a * 4 → a << 2`
  - `a * 7 → (a << 3) - a`
  - `a / 32767 → (a >> 15) + (a >> 30)`

    - Take advantage of mathematically sound simplification rules
    - Must be careful with floating-point and integer arithmetic (due to rounding and overflow/underflow)
    - Iteration of these optimizations is useful, but by how much

- Constant propagation

```
int x = 5;
int y = x * 2;  →  int y = 5 * 2;  →  int y = 10;  →
int z = a[y];      int z = a[y];      int z = a[y];    int z = a[10];
```

    - if a variable's value is a constant, replace its uses by the constant
    - value of a variable is propagated forward from the point of assignment (substitution operation)
    - to be most effective, constant propagation and folding interleave

- Copy propagation

    - If variable `y` is assigned to `x`, replace `x`'s uses with `y`
    - can make the first assignment to `x` dead code, thus eliminated

```
x = y;                          ~~x = y;~~
if (x > 1) {          →         if (y > 1) {
  x = x * f(x - 1);               x = y * f(y - 1);
}                               }
```

- Dead code elimination

```
x  = y * y   // x is dead!
...          // x never used  →      ...
x = z * z                           x = z * z
```

    - If side-effect free code can never be observed, safe to eliminated it

        - A variable is **dead** if it's never used after it is defined
            - computing such `def/use` information is an important compiler component
        - Dead variables can be created by other optimizations
- Unreachable/Dead code
        - Basic blocks unreachable from the entry block can be deleted
            - performed at the IR or assembly level, improves cache, TLB performance
        - Dead code: similar to unreachable blocks
            - a value might be computed but never subsequently used
            - Code for computing the value can be dropped
        - but only if it's pure (has no externally visible side effects)
            - Externally visible effects: raising an exception, modifying a global variable, going into an infinite loop, printing to standard output, sending a network packet, launching a rocket
            - Pure functional language make reasoning about the safety of optimizations (and code transformations in general) easier
- Inlining
        - Replace a function call with the body of the function (with arguments rewritten to be local variables)

```
int g(int x) { return x + pow(x); }
int pow(int a) { int b = 1; int n = 0;
   while (n < a) {b = 2 * b; n = n+1;}; return b; }
➔
int g(int x) { int a = x; int b = 1; int n = 0;
    while (n < a) {b = 2 * b; n = n+1;}; tmp = b;
    return x + tmp;
}
```

        - May need to rename variable names to avoid name capture
        - Best done at the AST or relatively high-level IR
        - Advantage: eliminates the stack manipulation, jump; enables further optimizations
        - Disadvantage: can increase code size
- Code specialization

- Idea: create specialized versions of a function that is called from different places with different arguments

```
class A implements I { int m() {…} }
class B implements I { int m() {…} }
int f(I x) { x.m(); }        // don't know which m
A a = new A(); f(a);         // know it's A.m
B b = new B(); f(b);         // know it's B.m
```

  - `f_A` would have code specialized to dispatch to `A.m`
  - `f_B` would have code specialized to dispatch to `B.m`
  - One can also inline methods when the runtime type is known statically
    - often just one class implements a method
    - through "Receiver Class Analysis" (RCA)

- Common Subexpression Elimination (CSE)
  - In some sense, CSE is the opposite of inlining: fold redundant computations together

```
a[i] = a[i] + 1   compiles to
[a + i*4] = [a + i*4] + 1
```

CSE removes the redundant add and multiply

```
t = a + i*4; [t] = [t] + 1
```

  - Safe: the shared expression must always have the same value in both places

- Unsafe CSE
  - Example: consider this OAT function

```
unit f(int[] a, int[] b, int[] c) {
  int j = …; int i = …; int k = …;
  b[j] = a[i] + 1;
  c[k] = a[i];
  return;
}
```

  - The following optimization that shares expression `a[i]` is unsafe, why?

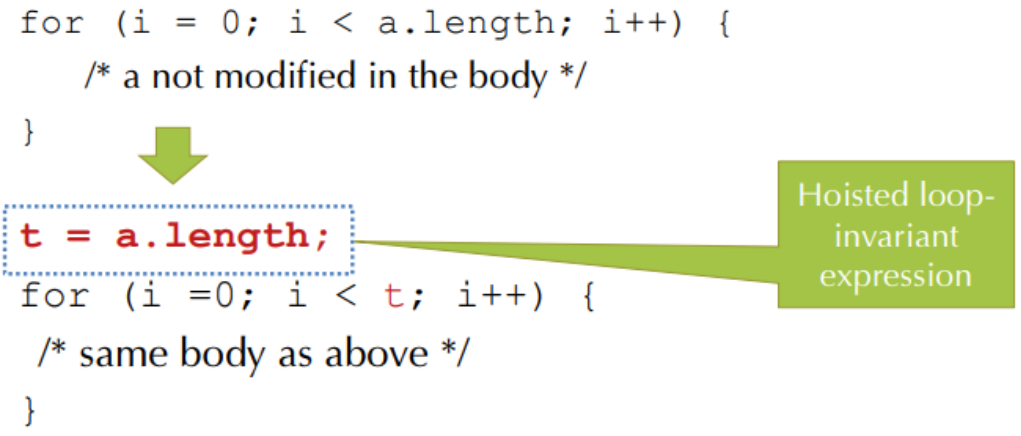
```
unit f(int[] a, int[] b, int[] c) {
  int j = …; int i = …; int k = …;
  t = a[i];
  b[j] = t + 1;
  c[k] = t;
  return;
}
```

- `a, b, c` can be the same array aliased (e.g. `b[j] == a[i]`)

Nov. 18th - Exercise Session 5: First-Class Functions, Closure Conversions and Types, and Subtyping, OO

Nov. 20th - Lecture 19: Optimization II, Data Flow Analysis, Cont.d; Register Allocation

- Loop Optimization
    - Program **hot spots** often occur in loops
        - especially inner loops
    - Most program execution time occurs in loops
        - 90% of the execution time is spent in 10% of the code
    - Loop optimizations are very important, effective, and numerous
    - Loop Invariant Code Motion (revisited)
        - Redundancy elimination: If the result of a statement or expression does not change during the loop and it's pure, it can be hoisted outside the loop body
        - Often useful for array element addressing code (invariant code not visible at the source level)



    - Strength Reduction (revisited)
        - Strength reduction: replace expensive operations by cheap ones

```
for (int i = 0; i<n; i++) { a[i*3] = 1; } // stride by 3
```

```
int j = 0;
for (int i = 0; i<n; i++) {
  a[j] = 1;
  j = j + 3;   // replace multiply by add
}
```

- Loop Unrolling (revisited)
    - With k unrollings, eliminates $\frac{k-1}{k}$ conditional branches
    - Space-time tradeoff
    - Interacts with instruction caching, branch prediction

Branches can be expensive, unroll loops to avoid them

```
for (int i=0; i<n; i++) { S }
```

```
int i;
for (i=0; i<n-3; i+=4) {S;S;S;S};
for (       ; i<n; i++) { S }    // left over iterations
```

- Effectiveness
    - Optimization Effectiveness
        - $\%speedup = (\frac{\text{base time}}{\text{optimized time}} - 1) \times 100\%$
        - **`mem2reg`**: promotes alloca'ed stack slots to temporaries to enable register allocation

Analysis
- mem2reg alone (+ back-end optimizations like register allocation) yields ~78% speedup on average
- -O1 yields ~100% speedup
  (so all the rest of the optimizations combined account for ~22%)
- -O3 yields ~120% speedup

Hypothetical program that takes 10 sec. (base time):
- Mem2reg alone: expect ~5.6 sec
- -O1: expect ~5 sec
- -O3: expect ~4.5 sec

- Code Analysis
    - Motivating Code Analyses

Many things might influence the safety/applicability of an optimization
- What algorithms and data structures can help?

How do we know what is a loop?
How do we know if an expression is invariant?
How do we know if an expression has no side effects?
How do we keep track of where a variable is defined?
How do we know where a variable is used?
How do we know if two reference values may be aliases of one another?

- Moving Toward Register Allocation
    - The OAT compiler current generates as many `temp` variables as it needs (these are `%uid`)
    - Current strategy
        - Each `%uid` maps to a stack location
        - This yields programs with many loads/stores to memory
    - Ideally, we would like to map as many `%uid` as possible into registers
        - eliminate the use of the `alloca` instruction
        - Only 16 max registers available on 64-bit X86
        - `%rsp, %rbp` reserved; some have special semantics, so only 10-12 available
        - This means that a register must hold more than one slot
- Liveness
    - Observation: `%uid1` and `%uid2` can be assigned to the same register if their values will not be needed at the same time
    - `%uid` is "needed" if its contents will be used as a source operand in a later instruction
    - Such a variable is called "live"
    - Two variables can share a register if they are not live at the same time
- Scope vs. Liveness
    - We can already get some coarse liveness information from variable scoping

- Consider

```
int f(int x) {
  var a = 0;
  if (x > 0) {
    var b = x * x;
    a = b + b;
  }
  var c = a * x;
  return c;
}
```

  - Due to OAT's scoping rules, `b` and `c` can never be live at the same time (`c`'s scope is disjoint from `b`'s scope)
  - So can assign `b, c` to the same alloca'ed slot & potentially to same register
- But Scope is too Coarse

```
int f(int x) {
  int a = x + 2;
  int b = a * a;
  int c = b + x;
  return c;
}
```

x is live
a and x are live
b and x are live
c is live

  - The scopes of `a, b, c, x` all overlap, all in scope at the end of the block
  - But they are never live at the same time, so they can share the same stack slot/register
- Live Variable Analysis
  - Variable `v` is live at a program point `L` if
    - `v` is defined before `L`
    - `v` is used after `L`
  - Liveness is defined in terms of where variables are defined and used
  - Liveness analysis: Compute the live variables between each statement
    - May be conservative (i.e. may claim a variable live when it is not) because that is a safe approximation

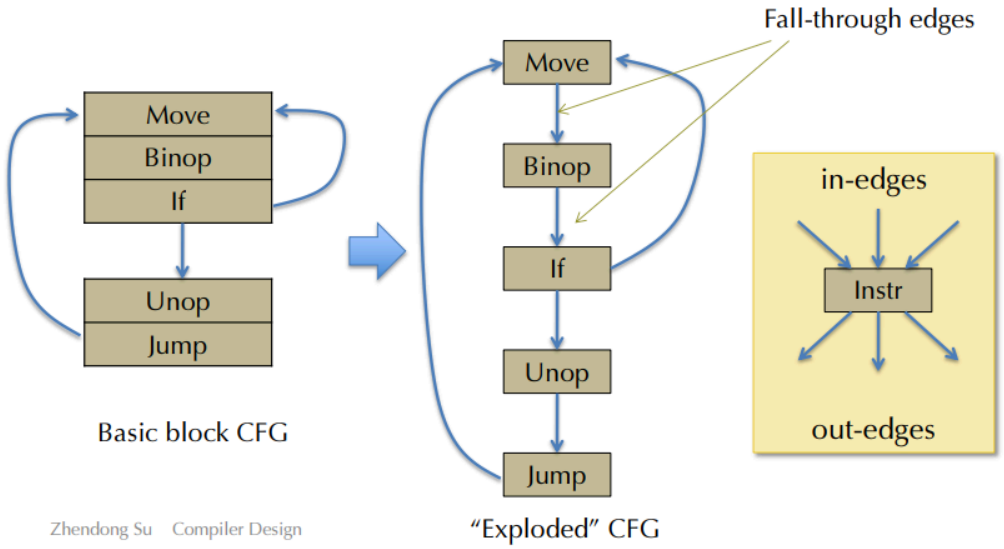
        - To be useful, it should be more precise than simple scoping rules
    - It is an example of dataflow analysis
- Control-flow Graphs Revisited

For dataflow analysis, we use the control-flow graph (CFG) intermediate form
Recall that a basic block is a sequence of instructions such that
- There is a distinguished, labeled entry point (no jumps into the middle of a basic block)
- There is a (possibly empty) sequence of non-control-flow instructions
- A block ends with a single control-flow instruction: jump, branch, return, etc.

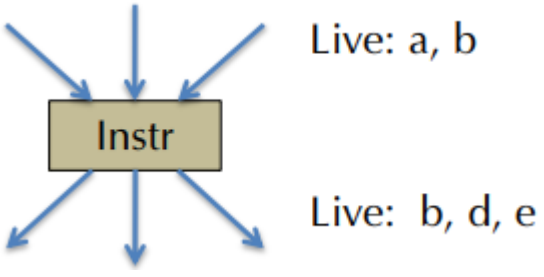
A control flow graph
- Nodes are blocks
- An edge from B1 to B2 if B1's control-flow instruction may jump to the entry label of B2
- There are no "dangling" edges – there is a block for every jump target

Note: the following slides are intentionally a bit ambiguous about the exact nature of the code in the control flow graphs
- at the x86 assembly level
- an "imperative" C-like source level
- at the LLVM IR level
- Same general idea, but the exact details will differ
    - e.g. LLVM IR doesn't have "imperative" update of %uid temporaries
    - In fact, the SSA structure of the LLVM IR makes some of these analyses simpler

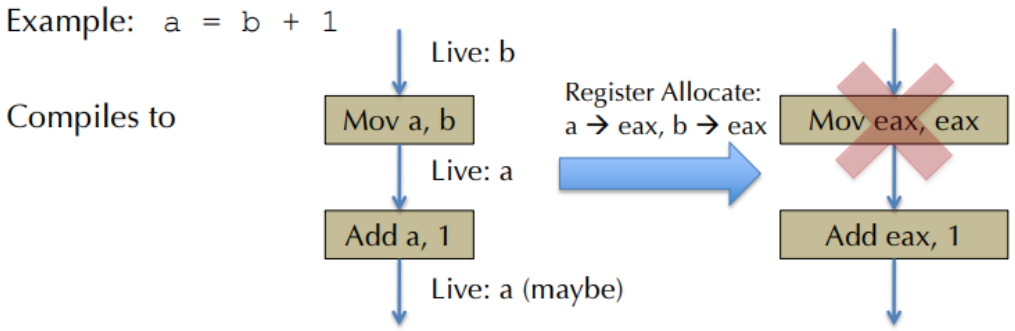
- Dataflow over CFGs
    - For precision, it is helpful to think of the "fall through" between sequential instructions as an edge of the control-flow graph
    - Different implementation tradeoffs in practice



- Liveness is associated with edges



- This is useful as the same register can be used for different temporaries in the same statement



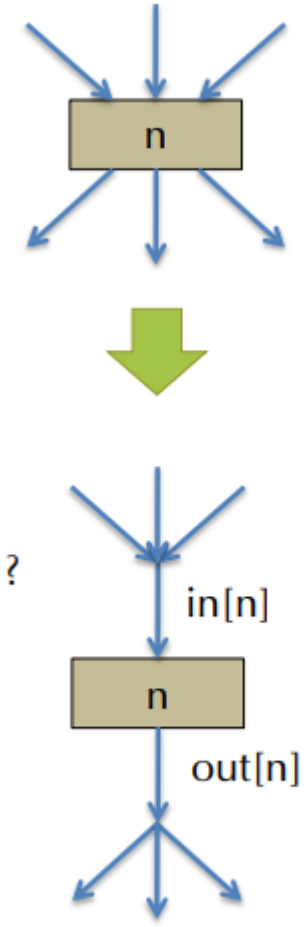
- Uses and Definitions
  - Every instruction/statement uses some set of variables (reads from them)
  - Every instruction/statement defines some set of variables (writes to them)
  - For a node/statement s define
    - `use`$[s]$: set of variables used by s
    - `def`$[s]$: set of variables defined by s

Examples:

– a = b + c	use[s] = {b,c}	def[s] = {a}
– a = a + 1	use[s] = {a}	def[s] = {a}

- Liveness, formally
  - A variable `v` is live on edge e if there is
    - a node n in the CFG such that `use`$[n]$ contains `v`, and
    - a directed path from e to n such that for every statement s' on the path, `def`$[s']$ does not contain `v`
  - The first clause says `v` will be used on some path starting from edge e
  - The second says that `v` won't be redefined on that path before the use
- Simple, inefficient algorithm
  - Backtracking algorithm
    - For each variable `v`, try all paths from each use of `v`, tracing backward through the control-flow graph until either `v` is defined or a previously visited node is reached, mark the variable `v` live across each edge traversed

        - Inefficient because it explores the same paths many times
- Dataflow Analysis
    - Idea: compute liveness information for all variables simultaneously - keep track of sets of information about each node
    - Approach: define equations that must hold by any liveness determination - equations based on "obvious" constraints
    - Solve the equations by iteratively converging on a solution
        - Start with a "rough" approximation to the answer
        - Refine the answer at each iteration
        - Keep going until no more refinement possible: a fixpoint has been reached
    - This is an instance of a general framework for computing program properties - dataflow analysis
- Dataflow Value Sets for Liveness
    - Nodes are program statements
        - `use`$[n]$: set of variables used by n
        - `def`$[n]$: set of variables defined by n
        - `in`$[n]$: set of variables live on entry to n
        - `out`$[n]$: set of variables live on exit from n
    - Associate `in`$[n]$ and `out`$[n]$ with the "collected" information about incoming/outgoing edges
    - For liveness: we obviously have constraint `out`$[n] \subseteq$ `in`$[n]$



- Other Dataflow Constraints
    - We have: a variable must be live on entry to n if it is used by n
        - $\texttt{use}[n] \subseteq \texttt{in}[n]$
    - If a variable is live on exit from n, and n does not define it, it is live on entry to n
        - $\texttt{out}[n] - \texttt{def}[n] \subseteq \texttt{in}[n]$
    - If a variable is live on entry to a successor node of n, it must be live on exit from n
        - if $n' \in \texttt{succ}[n]$, then $\texttt{in}[n'] \subseteq \texttt{out}[n]$
- Iterative Dataflow Analysis
    - Find a solution to those constraints by starting from a rough guess
    - Start with: $\texttt{in}[n] = \varnothing$ and $\texttt{out}[n] = \varnothing$
    - Idea: iteratively re-compute $\texttt{in}[n]$ & $\texttt{out}[n]$ where forced to by constraints - each iteration will add variables to the sets $\texttt{in}[n]$ and $\texttt{out}[n]$
    - We stop when the two sets satisfy these following equations derived from the constraints above
        - $\texttt{in}[n] = \texttt{use}[n] \cup (\texttt{out}[n] - \texttt{def}[n])$

- $\texttt{out}[n] = \bigcup_{n' \in \texttt{succ}[n]} \texttt{in}[n']$

- Complete Liveness Analysis Algorithm

```
for all n, in[n] := ∅, out[n] := ∅
repeat until no change in 'in' and 'out'
    for all n
        out[n] := ⋃_{n'∈succ[n]} in[n']
        in[n] := use[n] ∪ (out[n] \ def[n])
    end
end
```

  - finds a fixpoint of the `in` and `out` equations
    - the algorithm is guaranteed to terminate because the maximum is all the variables, and the two sets only increase in size
  - Why do we start with ∅?
    - To compute the least possible solution for the two sets

- Improving the Algorithm
  - Observe: information propagates between nodes only via $\texttt{out}[n]$ updating - this is the only rule that involves more than one node
  - If a node n's successors have not changed, n won't change
  - Thus, idea for an improved version of the algorithm involves keeping track of which node's successors have changed

- A worklist algorithm

Use a FIFO queue of nodes that might need to be updated

```
for all n, in[n] := ∅, out[n] := ∅
w = new queue with all nodes
repeat until w is empty
    let n = w.pop()                    // pull a node off the queue
     old_in = in[n]                    // remember old in[n]
     out[n] := ⋃_{n'∈succ[n]} in[n']
     in[n] := use[n] ∪ (out[n] - def[n])
     if (old_in != in[n]),             // if in[n] has changed
       for all m in pred[n], w.push(m) // add to worklist
end
```

- Register Allocation
    - Register Allocation Problem
        - Given: an IR program using an unbounded number of temporaries
        - Find: a mapping from temporaries to machine registers such that
            - program semantics is preserved
            - register usage is maximized
            - moves between registers are minimized
            - calling conventions / architecture requirements are obeyed
        - Stack Spilling
            - if $\exists$ k registers available and $m > k$ temps live at the same time, not all temps will fit into registers
            - So "spill" the excess temps to the stack
    - Linear-Scan Register Allocation
        - Simple, greedy register-allocation strategy
        - 1. Compute liveness information: `live(x)`
             `live(x)`: the set of uids that are live on entry to `x`'s definition
          2. Let `pal` be the set of usable registers
             Usually reserve a couple for spill code
             (our implementation uses rax, rcx)
          3. Maintain "layout" `uid_loc` that maps uids to locations
             Locations include registers and stack slots n, starting at n=0
          4. Scan through the program, for each instruction that defines a uid `x`

```
used = {r | reg r = uid_loc(y) s.t. y ∈ live(x)}
available = pal - used
If available is empty:              // no registers available, spill
  uid_loc(x) := slot n  ; n = n + 1
Otherwise, pick r in available:     // choose an available register
  uid_loc(x) := reg r
```

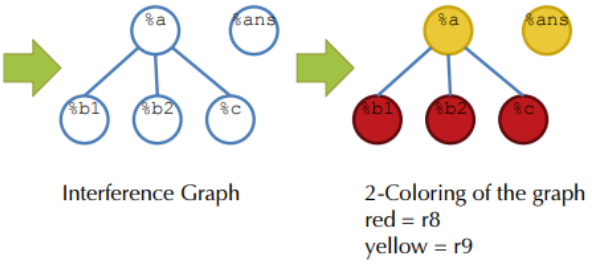
- Graph Coloring
    - Register Allocation

Basic process

1. **Compute liveness** information for each temp
2. Create an **interference graph**
   - Nodes are temps
   - There is an edge between nodes n & m if they are **live at the same time**
3. Try to **color the graph**
   - Each color corresponds to a register
4. If step 3 fails, **"spill" a register** to the stack and repeat from step 1
5. Rewrite the program to use registers

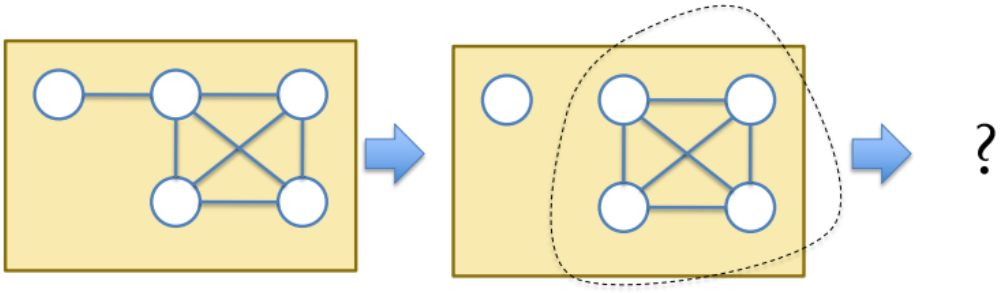
- Interference Graphs
  - Nodes of the graph are `%uid`
  - Edges connect variables that interfere with each other
    - two variables interfere if their live ranges intersect (i.e. there is an edge in the control-flow graph across which they are both live)
  - Register assignment is a graph coloring
    - A graph coloring assigns each node in the graph a color (register)
    - any two nodes connected by an edge must have different colors

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1
// live = {%a,%c}
%b2 = add i32 %c, 1
// live = {%a,%b2}
%ans = mult i32 %b2, %a
// live = {%ans}
return %ans;
```



- Questions
  - Can we efficiently kind a k-coloring of the graph whenever possible?
    - In general the problem is NP-complete, but we can do an efficient approximation using heuristics
  - How do we assign registers to colors
    - if done in a smart way, we can eliminate redundant MOVs
  - What do we do when there are not enough colors/registers
    - have to use stack space
- Coloring a Graph: Kempe's Algorithm

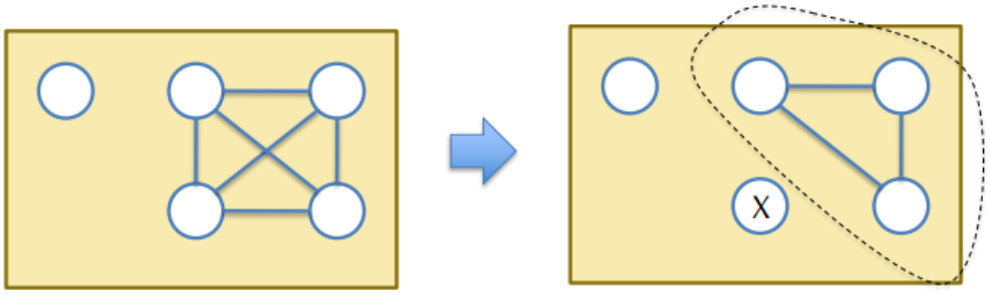
        - Find a node with degree $< K$ and cut it out of the graph
            - remove the nodes and edges
            - this is called simplifying the graph
        - Recursively K-color the remaining subgraph
        - When remaining graph is colored, there must be at least one free color available for the deleted node, pick such color
    - Failure of the Algorithm
        - If the graph cannot be colored, it will simplify to a graph where every node having $\geq K$ neighbours
            - this can happen even when the graph is K-colorable
            - symptom of NP-hardness



Nov. 22nd - Lecture 20: Register Allocation & Data Flow Analysis

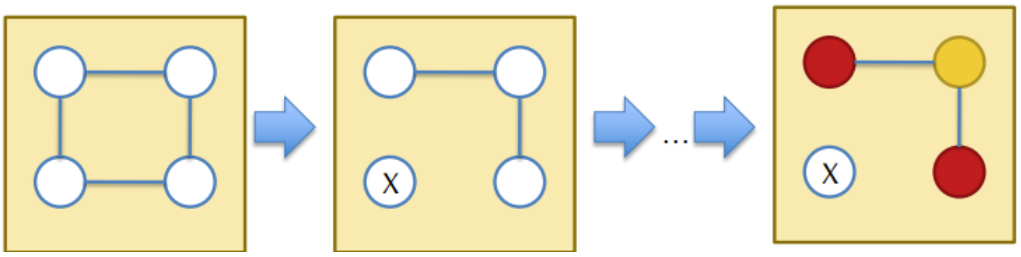
- Register Allocation
    - Spilling
        - Idea: if we cannot k-color the graph, we need to store 1 temp on the stack
        - Which variable to spill
            - pick one that is not used very frequently
            - pick one that is not used in a (deeply nested) loop
            - pick one that has high interference (since removing it will make the graph easier to color)
        - In practice: some weighted combination of these criteria
        - When coloring
            - Mark the node as spilled
            - Remove it from the graph
            - Keep recursively coloring
    - Spilling pictorially

- Select a node to spill, mark it and remove it from the graph, continue coloring



- Optimistic Coloring
    - Sometimes it is possible to color a node marked for spilling
        - if we get "lucky" with the choices of colors made earlier

Example: When 2-coloring this graph



    - Even though the node was marked for spilling, we can color it
    - So, on the way down, mark for spilling, but don't need to actually spill
- Accessing Spilled Registers
    - if optimistic coloring fails, we need to generate code to move the spilled temps to/from memory
    - Option 1: Rewrite the program to use a new temp with explicit moves to/from memory
        - Pro: need to reserve fewer registers
        - Con: introducing temporaries changes live ranges, so must recompute liveness & recolor graph
    - Option 2: Reserve registers specifically for moving to/from memory
        - Pro: only need to color the graph once
        - Con: need at least 2 registers (one for each source operand of an instruction), so decreases total number of available registers by 2

- Not good on x86 (especially 32 bit) because there are too few registers & too many constraints on how they can be used

- Option 1

Suppose temp `t` is marked for spilling to stack slot `[rbp+offs]`
Rewrite the program like this

```
t = a op b;          t = a op b          // defn. of t
                     Mov [rbp+offs], t

x = t op c           Mov t37, [rbp+offs] // use #1 of t
                     x = t37 op c

y = d op t           Mov t38, [rbp+offs] // use #2 of t
                     y = d op t38
```

`t37` & `t38` are fresh temps to replace `t` for its different uses
Rewriting the code in this way breaks `t`'s live range up:
`t, t37, t38` are only live across one edge

- Option 2

Suppose temp `t` is marked for spilling to stack slot `[rbp+offs]`
Rewrite the program like this

```
t = a op b;          t = a op b          // defn. of t
                     Mov [rbp+offs], t

x = t op c           Mov r, [rbp+offs]   // use #1 of t
                     x = r op c

y = d op t           Mov r, [rbp+offs]   // use #2 of t
                     y = d op r

u = u op v;          Mov r1, [rbp+offs1] // both u, v spilled
                     Mov r2, [rbp+offs2] // u@offs1, v@ offs2
                     r1 = r1 op r2
                     Mov [rbp+offs1], r1
```

- Precolored Nodes
  - Some variables must be **pre-assigned** to registers
    - on x86 the multiplication instruction: `IMul` must define `%rax`
    - The "Call" instruction should kill caller-save registers `%rax, %rcx, %rdx`
    - Any temp live across a call interferes with the caller-save registers
  - To properly allocate temps, we treat registers as nodes in the interference graph with pre-assigned colors

        - pre-colored nodes cannot be removed during simplification
        - Trick: treat pre-colored nodes as having "infinite" degree in the interference graph to guarantee they won't be simplified
        - when the graph is empty except the pre-colored nodes, we have reached the point where we start coloring the rest of the nodes
- Picking good colors
    - When choosing colors during the coloring phase, any choice is semantically correct, but some choices are better for performance

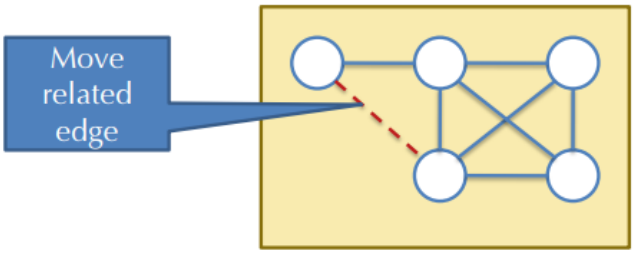
Example: `movq t1, t2` → `movq r, r`

– If t1 and t2 can be assigned the same register (color) `r`, this move is redundant and can be eliminated

    - A simple color choosing strategy that helps eliminate such moves
        - add a new kind of "move related" edge between the nodes for `t1` and `t2` in the interference graph
        - when choosing a color for `t1` (or `t2`), if possible pick a color of an already colored node reachable by a move-related edge

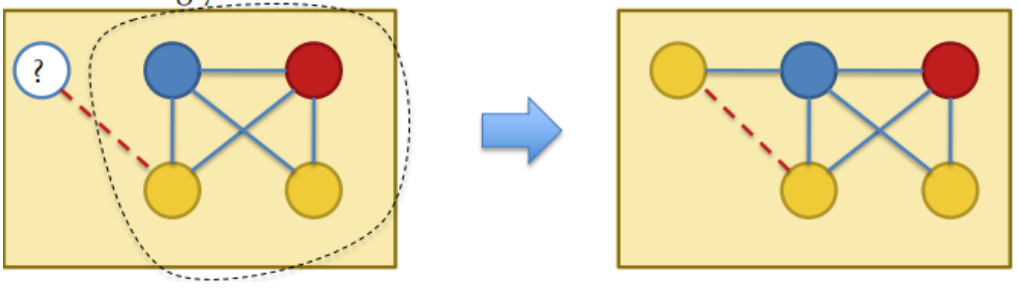
- Example Color choice

Consider 3-coloring this graph, where the dashed edge indicates that there is a Mov from one temp to another



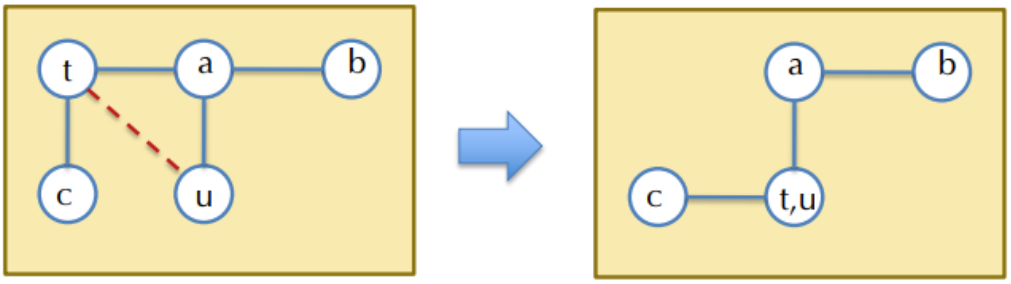
After coloring the rest, we have a choice

– Picking yellow is better than red because it will eliminate a move

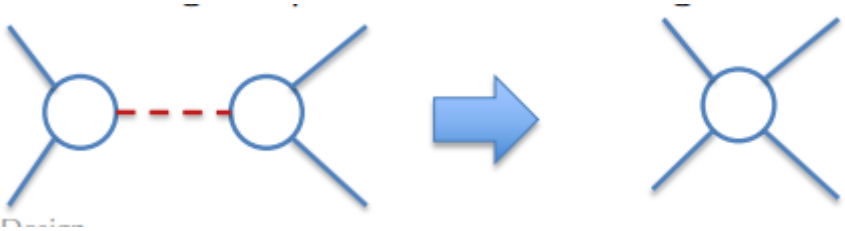


- Coalescing Interference Graphs

- a more aggressive strategy is to **coalesce** nodes of the interference graph if they are connected by move-related edges
    - coalescing nodes **forces** the two temps to be assigned the same register



- Idea: interleave simplification and coalescing to maximize the number of moves that can be eliminated
- Problem: coalescing may increase the degree of a node

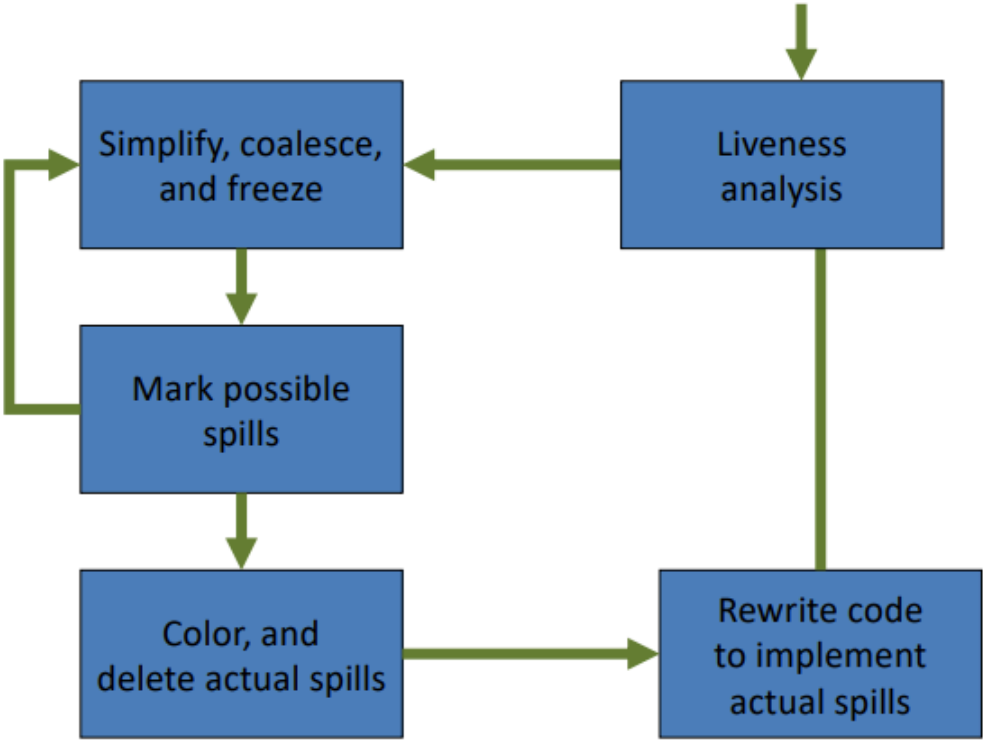


- Conservative Coalescing
    - guarantee to preserve k-colorability of interference graphs
    - Briggs' strategy: it is safe to coalesce x & y if the resulting node will have fewer than k neighbors that have degree $\geq k$
        - The merged node (x, y) can still be removed
    - George's strategy: can safely coalesce x & y if for every neighbor t of x, either t already interferes with y or t has degree $< k$
        - Let S be the set of neighbors of x with degree $< k$
        - if no coalescing, simplify and remove all nodes in $S \rightarrow G_1$
        - if we coalesce, still can remove all nodes in $S \rightarrow G_2$
        - G_2 is a subgraph of G_1
- Why 2 strategies?
    - Briggs: we need to look at all neighbors of x & y
    - George: we need to look at only the neighbors of x
    - Pre-colored nodes have infinite degree
    - Thus, we use Briggs if both are temporaries, use George if one of x & y is pre-colored

- Complete Register Allocation Algorithm

1. Build interference graph (precolor nodes as necessary)
   - Add move related edges
2. Reduce the graph (building a stack of nodes to color)
   1. Simplify the graph as much as possible without removing nodes that are move related (i.e. have a move-related neighbor). Remaining nodes are high degree or move-related
   2. Coalesce move-related nodes using Briggs' or George's strategy
   3. Coalescing can reveal more nodes that can be simplified, so repeat 2.1 and 2.2 until no node can be simplified or coalesced
   4. If no nodes can be coalesced, **freeze** (remove) a move-related edge and keep trying to simplify/coalesce
3. If there are non-precolored nodes left, mark one for spilling, remove it from the graph and continue doing step 2
4. When only pre-colored node remain, start coloring (popping simplified nodes off the top of the stack)
   1. If a node must be spilled, insert spill code as shown earlier and rerun the whole register allocation algorithm starting at step 1

- Overall Algorithm



- Last Details
  - After register allocation, the compiler should do a peephole optimization pass to remove redundant moves
  - Some architectures specify calling conventions that use registers to pass function arguments
    - it's helpful to move such arguments into temps in the function prelude so that compiler has as such freedom as possible during register allocation
    - not an issue on x86

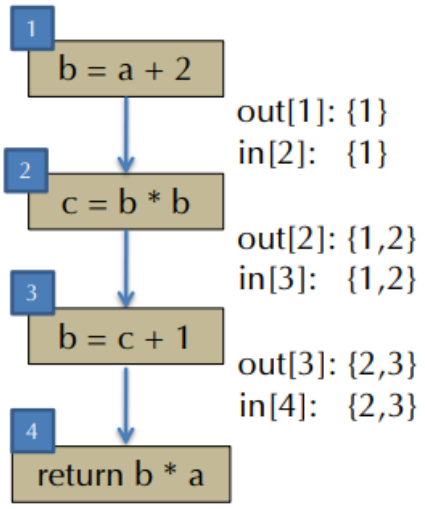
        - Vast literature on register allocation
        - Other notable formulations include using Integer Linear Programming (for optimal assignment)
- Other Dataflow Analyses
    - Generalizing Dataflow Analysis
        - This type of iterative analysis for liveness also applies to other analyses
            - reaching definitions analysis
            - available expressions analysis
            - alias analysis
            - constant propagation
            - these analyses follow the same 3-step approach as for liveness
        - The examples work over a canonical IR called **quadruples**
            - allows easy definition of `def`$[n]$ and `use`$[n]$
            - slightly "looser" LLVM IR variant that does not require SSA (i.e. it has mutable local variables)
            - we will use LLVM-IR-like syntax
    - Def / Use for SSA

Instructions n	def[n]	use[n]	description
a = op b c	{a}	{b,c}	arithmetic
a = load b	{a}	{b}	load
store b, c	Ø	{b}	store
a = alloca t	{a}	Ø	alloca
a = bitcast b to u	{a}	{b}	bitcast
a = gep b [c,d, …]	{a}	{b,c,d,…}	getelementptr
a = f(b_1,…,b_n)	{a}	{b_1,…,b_n}	call w/return
f(b_1,…,b_n)	Ø	{b_1,…,b_n}	void call (no return)
Terminators			
br L	Ø	Ø	jump
br a L1 L2	Ø	{a}	conditional branch
return a	Ø	{a}	return

- Reaching Definitions
    - Reaching Definition Analysis
        - Q: What variable definitions reach a particular use of the variable?
        - This analysis is used for constant propagation & copy propagation

        - Constant propagation: if only one definition reaches a particular use, can replace use by the definition
        - Copy propagation: additionally requires that the copied value still has its same value - computed using an available expressions analysis
    - Input: Quadruple + CFG
    - Output: `in`$[n]$ (resp. `out`$[n]$) is the set of nodes defining some variable such that the definition may reach the beginning (resp. end) of node n
- Example

Results of computing reaching definitions on this simple CFG



- Step 1
    - Define the sets of interest for the analysis
    - Let `defs`$[a]$ be the set of nodes that define the variable a
    - Define `gen`$[n]$ and `kill`$[n]$ as follows

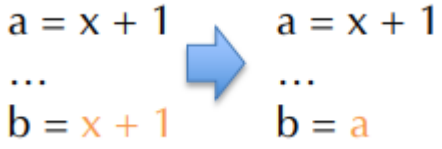
Quadruple forms n	gen[n]	kill[n]
a = b op c	{n}	defs[a] \ {n}
a = load b	{n}	defs[a] \ {n}
store b, a	∅	∅
a = f(b_1,…,b_n)	{n}	defs[a] \ {n}
f(b_1,…,b_n)	∅	∅
br L	∅	∅
br a L1 L2	∅	∅
return a	∅	∅

    - `kill`$[n]$ is the set of nodes that defines a apart from node n, because instruction n re-defines a, all other definitions are "killed"
- Step 2

        - Define the constraints that a reaching definitions solution must satisfy
        - $\texttt{gen}[n] \subseteq \texttt{out}[n]$
            - definitions reaching the end of a node at least include the definitions generated by the node
        - $\texttt{out}[n'] \subseteq \texttt{in}[n]$, if $n' \in \texttt{pred}[n]$
            - definitions reaching the beginning of a node include those that reach the exit of **any** predecessor
        - $\texttt{in}[n] \subseteq \texttt{out}[n] \cup \texttt{kill}[n]$
            - definitions coming into a node n either reach the end of n or are killed by it
            - $\texttt{in}[n] - \texttt{kill}[n] \subseteq \texttt{out}[n]$
    - Step 3
        - Convert constraints to iterated update equations

          $\texttt{in}[n] := \bigcup_{n' \in \texttt{pred}[n]} \texttt{out}[n']$

          $\texttt{out}[n] := \texttt{gen}[n] \cup (\texttt{in}[n] - \texttt{kill}[n])$
        - Algorithm: initialize `in` and `out` to $\varnothing$, iterate the update equations until a fixed point is reached
        - Algorithm **terminates** since both sets increase only **monotonically**, at most to a maximum set that includes all variables in the program
        - It is **precise** since it finds the smallest sets that satisfy the constraints
- Available Expressions
    - Idea: want to perform common subexpression elimination

    - It is safe if $x + 1$ computes the same value at both places
        - "$x + 1$" is an available expression
    - Dataflow values
        - $\texttt{in}[n]$: set of nodes whose values are available on entry to n
        - $\texttt{out}[n]$: set of nodes whose values are available on exit of n
    - Step 1

Define the sets of values

Define gen[n] and kill[n] as follows

Quadruple forms n	**gen[n]**	**kill[n]**
a = b op c	{n} \ kill[n]	uses[a]
a = load b	{n} \ kill[n]	uses[a]
store b, a	Ø	uses[[x]] (for all x that may equal a)
br L	Ø	Ø
br a L1 L2	Ø	Ø
a = f(b_1,…,b_n)	Ø	uses[a]∪ uses[[x]] (for all x)
f(b_1,…,b_n)	Ø	uses[[x]] (for all x)
return a	Ø	Ø

Note the need for "may alias" information...

Note that functions are assumed to be impure

- **`gen`**: consider the expression `a = a + 1`, a is used in `a + 1` but the expression will be no longer available because we have updated a, so we need to rid the **`kill`**
- **`kill`**: we want the expressions that require a to be "killed"

- Step 2

Define constraints that an available expressions solution must satisfy

out[n] ⊇ gen[n]
"Expressions made available by n reach the end of the node"

in[n] ⊆ out[n'] if n' is in pred[n]
"Expressions available at the beginning of a node include only those that reach the exit of *every* predecessor"

out[n] ∪ kill[n] ⊇ in[n]
"Expressions available on entry either reach the end of the node or are killed"
– Equivalently: out[n] ⊇ in[n] \ kill[n]

Note similarities and differences with constraints for "reaching definitions".

- Step 3

Convert constraints to iterated update equations

$$in[n] := \bigcap_{n' \in pred[n]} out[n']$$
$$out[n] := gen[n] \cup (in[n] \setminus kill[n])$$

Algorithm: initialize in[n] and out[n] to the set of all nodes
- Iterate the update equations until a fixed point is reached

Algorithm **terminates** since in[n] & out[n] decrease only monotonically
- At most to a minimum of the empty set

It is **precise** since it finds the largest sets that satisfy the constraints

- General Dataflow Analyses
  - Look at the update equations in the inner loop of the analyses

Liveness **(backward, may)**
- Let gen[n] = use[n] and kill[n] = def[n]
- $out[n] := \bigcup_{n' \in succ[n]} in[n']$
- $in[n] := gen[n] \cup (out[n] \setminus kill[n])$

Reaching Definitions **(forward, may)**
- $in[n] := \bigcup_{n' \in pred[n]} out[n']$
- $out[n] := gen[n] \cup (in[n] \setminus kill[n])$

Available Expressions **(forward, must)**
- $in[n] := \bigcap_{n' \in pred[n]} out[n']$
- $out[n] := gen[n] \cup (in[n] \setminus kill[n])$

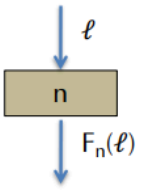
  - Very Busy Expressions
    - Expression e is very busy at location p if every path from p must evaluate e before any variable in e is redefined
    - Optimization: hoisting expressions
    - A must-analysis, a backward analysis
  - One cut at the dataflow design space

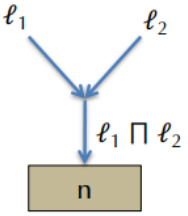
	May	Must
Forward	Reaching definitions	Available expressions
Backward	Live variables	Very busy expressions

- "Bidirectional analyses"
- Common features
    - All of these analyses have a **domain** over which they solve constraints
        - Liveness: domain = sets of variables
        - Other: domain = sets of nodes
    - Each analysis has a notion of `gen` and `kill`
        - used to explain how information propagates across a node
    - Each analysis propagates information either **forward** or **backward**
        - forward: `in` defined in terms of predecessor nodes' `out`
        - backward: `out` defined in terms of successor nodes' `in`
    - Each analysis has a way of aggregating information
        - union - may/intersection - must
- (Forward) Dataflow Analysis Framework

A forward dataflow analysis can be characterized by

1. A domain of dataflow values $\mathcal{L}$
   - e.g. $\mathcal{L}$ = the powerset of all variables
   - Think of $\ell \in \mathcal{L}$ as a property, then "x ∈ ℓ" means "x has the property"
2. For each node n, a flow function $F_n : \mathcal{L} \rightarrow \mathcal{L}$
   - So far we've seen $F_n(\ell) = gen[n] \cup (\ell \setminus kill[n])$
   - So: $out[n] = F_n(in[n])$
   - "If ℓ is a property that holds before the node n, then $F_n(\ell)$ holds after n"
3. A combining operator $\sqcap$
   - "If we know *either* ℓ_1 *or* ℓ_2 holds on entry to node n, we know at most $\ell_1 \sqcap \ell_2$"
   - $in[n] := \bigsqcap_{n' \in pred[n]} out[n']$



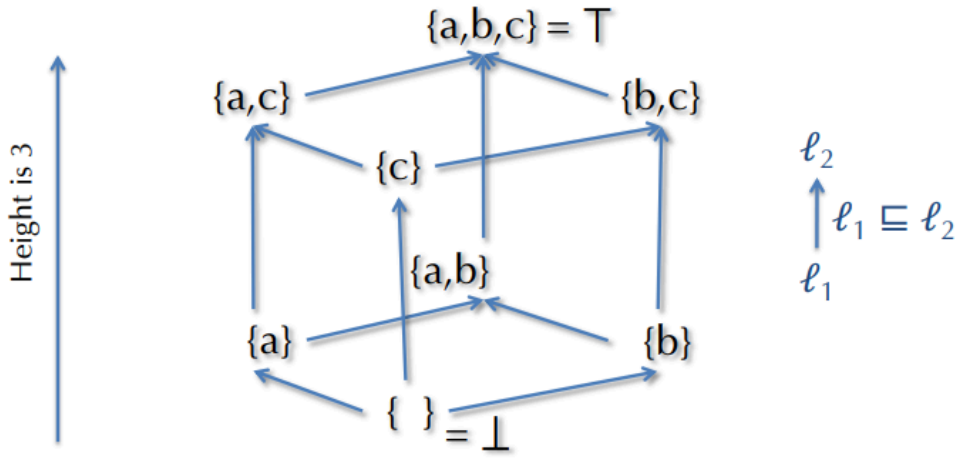


- Generic Iterative (Forward) Analysis

```
for all n, in[n] := ⊤, out[n] := ⊤
repeat until no change
    for all n
        in[n] := ⨅_{n'∈pred[n]} out[n']
        out[n] := F_n(in[n])
    end
end
```

- $\top \in \mathcal{L}$ ("top") represents having the "maximum" amount of information
  - Having "more" information enables more optimizations
  - "Maximum" amount could be inconsistent with the constraints
  - Iteration refines the answer, eliminating inconsistencies

- Structure of L
- L as a Partial Order
  - L is a **partial order** defined by the ordering relation
  - it is an ordered set
  - Properties: reflexivity, transitivity, anti-symmetry



order $\sqsubseteq$ is $\subseteq$ meet $\sqcap$ is $\cap$ join $\sqcup$ is $\cup$

- Meets and Joins
  - Meet - constructs the greatest lower bound
  - Join - constructs the least upper bound
  - A partial order that has all meets and joins is called a **lattice**
- Another way to describe the algorithm

Algorithm repeatedly computes (for each node n)

$out[n] := F_n(in[n])$

Equivalently: $out[n] := F_n(\bigsqcap_{n' \in pred[n]} out[n'])$

- By definition of in[n]

We can write this as a simultaneous update of the vector of out[n]

- let $x_n = out[n]$
- Let $\mathbf{X} = (x_1, x_2, \ldots, x_n)$ it's a vector of points in $\mathcal{L}$
- $\mathbf{F}(\mathbf{X}) = (F_1(\bigsqcap_{j \in pred[1]} out[j]), F_2(\bigsqcap_{j \in pred[2]} out[j]), \ldots, F_n(\bigsqcap_{j \in pred[n]} out[j]))$

Any solution to the constraints is a fixpoint **X** of **F**, i.e., **F**(**X**) = **X**

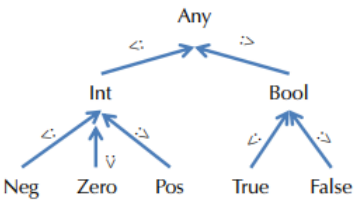
- Iteration Computes Fixpoints
- Monotonicity and Termination
- Building Lattices?

Information about individual nodes or variables can be lifted pointwise

- If $\mathcal{L}$ is a lattice, then so is $\{ f : X \to \mathcal{L} \}$ where $f \sqsubseteq g$ if and only if $f(x) \sqsubseteq g(x)$ for all $x \in X$

Like types, the dataflow lattices are static approx. to dynamic behavior

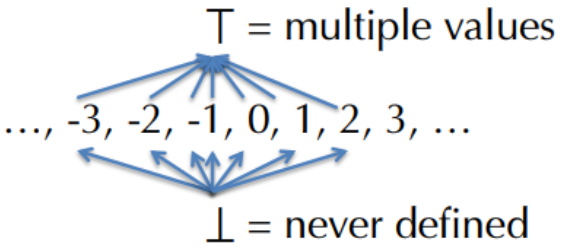
- Could pick a lattice based on subtyping
- Or other information



Points in the lattice are sometimes called dataflow "facts"

- "Classic" Constant Propagation
  - Constant propagation can be formulated as a dataflow analysis
  - Idea: propagate and fold integer constants in one pass
  - Information about a single variable
    - variable is never defined
    - variable has a single, constant value
    - variable is assigned multiple values
- Domains for Constant Propagation

We can make a constant propagation lattice $\mathcal{L}$ for one variable like



- Flow Functions

Consider the node x = y op z

$F(\ell_x, \ell_y, \ell_z) = ?$

$F(\ell_x, \top, \ell_z) = (\top, \top, \ell_z)$
$F(\ell_x, \ell_y, \top) = (\top, \ell_y, \top)$
"If either input might have multiple values the result of the operation might too."

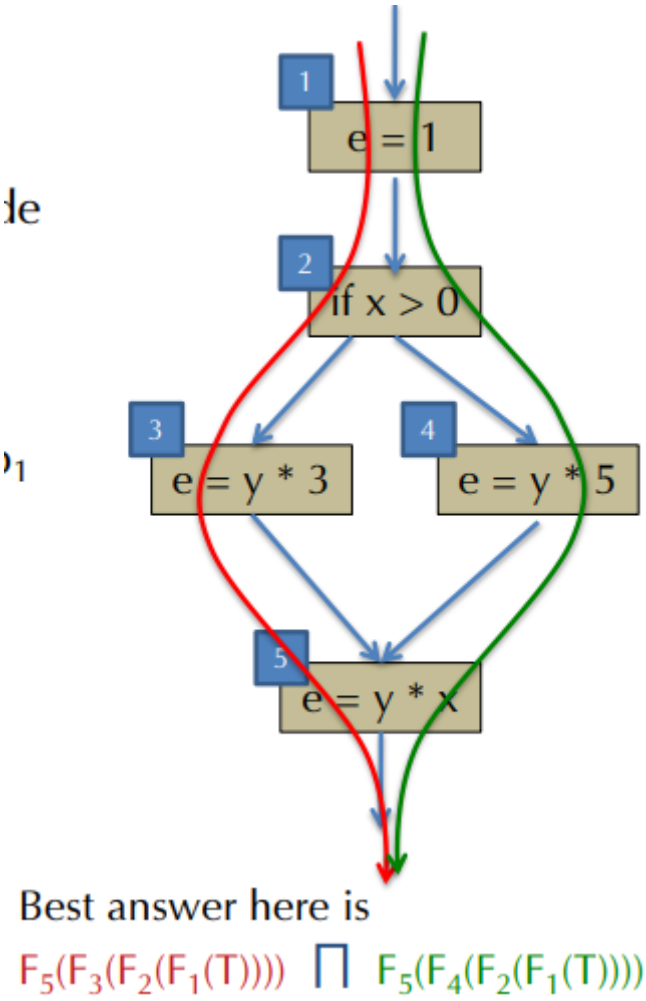
$F(\ell_x, \bot, \ell_z) = (\bot, \bot, \ell_z)$
$F(\ell_x, \ell_y, \bot) = (\bot, \ell_y, \bot)$
"If either input is undefined the result of the operation is too."

$F(\ell_x, i, j) = (i \text{ op } j, i, j)$
"If the inputs are known constants, calculate the output statically."

Flow functions for the other nodes are easy

Nov. 27th - Lecture 21: Data Flow Analysis II, Control Flow Analysis + SSA Revisited

- Quality of dataflow analysis solutions
    - Best Possible Solution



- Suppose we have a CFG
- If $\exists$ path p_1 starting from the root node (function entry) traversing the nodes $n_0, n_1, n_2, \ldots, n_k$
- The best possible information along p_1 is $l_{p_1} = F_{n_k}(\ldots F_{n_2}(F_{n_1}(F_{n_0}(T))))$
- Best solution at the output is some $l \subseteq l_p$ for all paths p
- Meet-over-paths (MOP) solution: $\bigcap_{p \in \text{paths-to}[n]} l_p$

- What about quality of iterative solutions?
  - Does the iterative solution compute the MOP solution?

    $\texttt{out}[n] = F_n(\bigcap_{n' \in \texttt{pred}[n]} \texttt{out}[n'])$ vs. $\bigcap_{p \in \text{paths-to}[n]} l_p$
  - Answer: yes, if the flow functions distribute over $\bigcap$
    - This means: $\bigcap_i F_n(l_i) = F_n(\bigcap_i l_i)$
    - Difficulty: loops in the CFG may mean there are infinitely many paths
  - Not all analyses give MOP solution (they are more conservative)
- Reaching Definition is MOP

$F_n[x] = gen[n] \cup (x \setminus kill[n])$

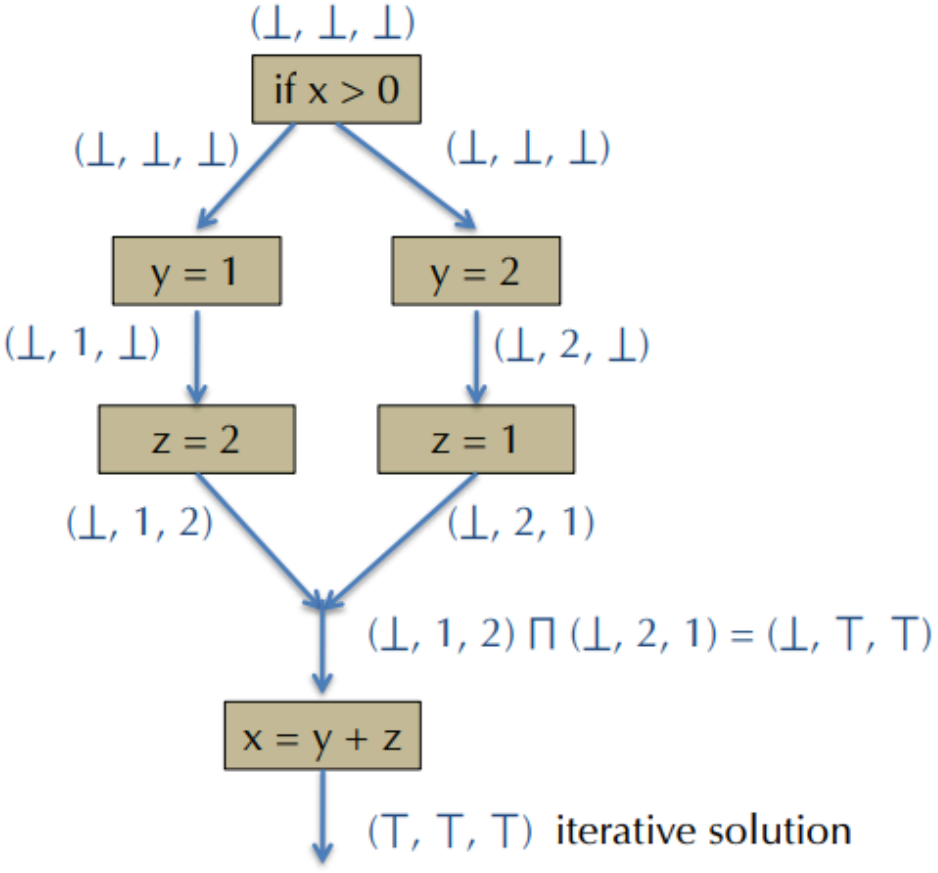
Does F_n distribute over meet $\sqcap = \cup$?

$F_n[x \sqcap y]$
$= gen[n] \cup ((x \cup y) \setminus kill[n])$
$= gen[n] \cup ((x \setminus kill[n]) \cup (y \setminus kill[n]))$
$= (gen[n] \cup (x \setminus kill[n])) \cup (gen[n] \cup (y \setminus kill[n])$
$= F_n[x] \cup F_n[y]$
$= F_n[x] \sqcap F_n[y]$

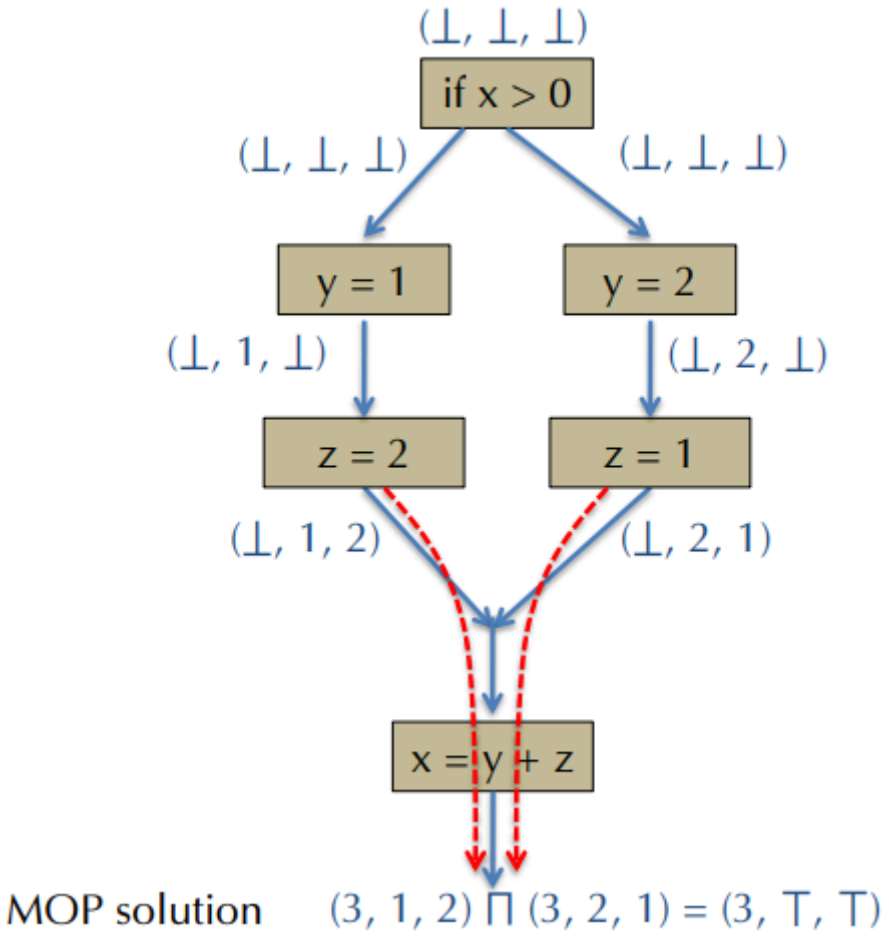
Therefore, Reaching Definitions with iterative analysis always terminates with the MOP (i.e., best) solution

In fact, the other three analyses (i.e., liveness, available expressions, & very busy expressions) are all MOP

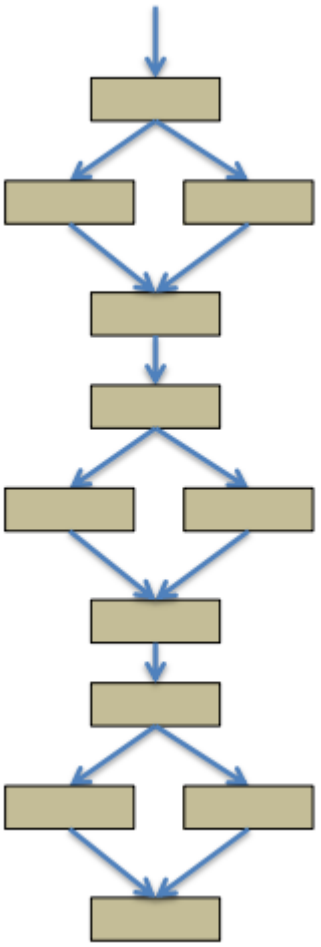
- Constant propagation iterative solution



- MOP Solution $\neq$ Iterative solution



- What problems are distributive?
    - Liveness analysis, available expressions, reaching definitions, very busy expressions
    - These express properties on how the program computes
- What problems are not distributive?
    - Analyses of what the program computes
        - the output is a constant, positive, and so on
        - constant propagation
- Why not compute MOP solution
    - If MOP is better than the iterative analysis, why don't we compute it? There are exponentially many paths
    - $\mathcal{O}(n)$ nodes, $\mathcal{O}(n)$ edges, and $\mathcal{O}(2^n)$ paths



- Terminology Review

Review

- Must vs. May
- Forward vs. Backward
- Distributive vs. non-Distributive

Additional

- Flow-sensitive vs. Flow-insensitive
- Context-sensitive vs. Context-insensitive
- Path-sensitive vs. Path-insensitive

- Summary

Many dataflow analyses fit into a common framework

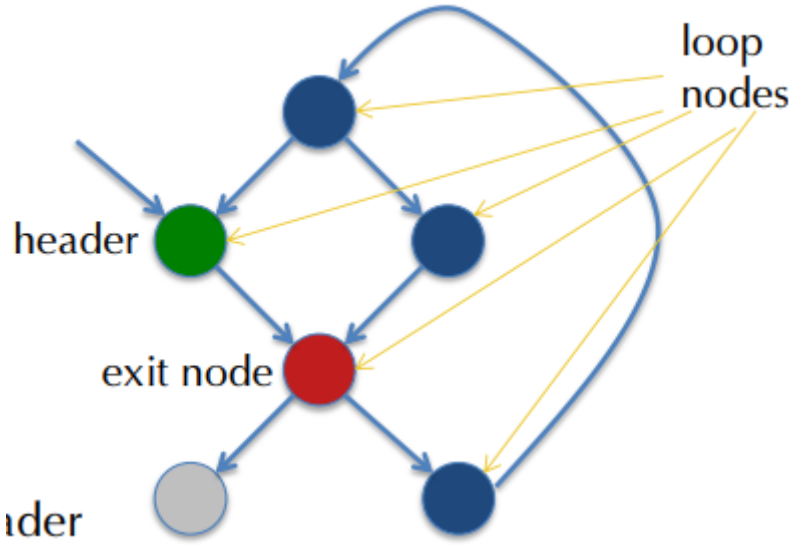
Key idea: **Iterative solution** of a system of equations over a **lattice**

- Iteration terminates if flow functions are monotonic
- Equivalent to the MOP answer if flow functions distribute over meet (⊓)

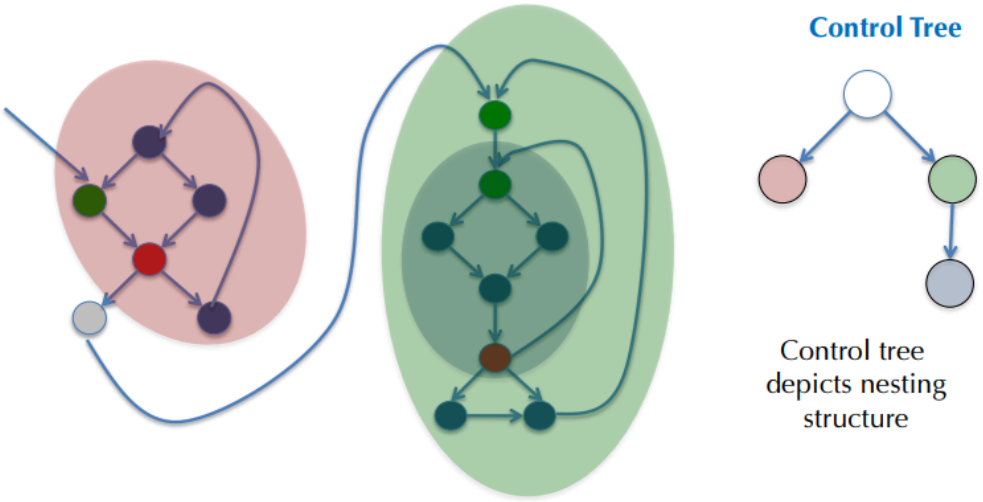
Dataflow analyses as presented work for an "imperative" IR

- Values of temporary variables are updated ("mutated") during evaluation
- Such mutation complicates calculations
- SSA = "Single Static Assignment" eliminates this problem
  - By introducing more temporaries --- each one assigned to only once

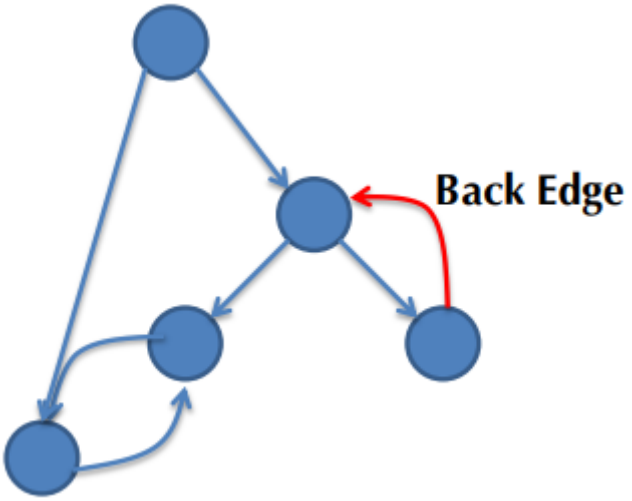
- Control-flow analysis & SSA
- Loops and Dominators
    - Loops in CFG
        - Optimizing loop bodies is important
        - Loop optimizations benefit from other IR-level optimizations and vice-versa (it's good to interleave them)
        - Loops may be hard to recognize at the quadruple / LLVM IR level
        - How do we identify loops in the CFG
    - Definition of a Loop
        - A loop is a set of nodes in the CFG
            - one distinguished entry: the header
        - Each node reachable from header
        - header reachable from each node
        - Loop is a strongly connected component
        - No edges enter a loop except to header
        - Exit nodes: nodes with outgoing edges



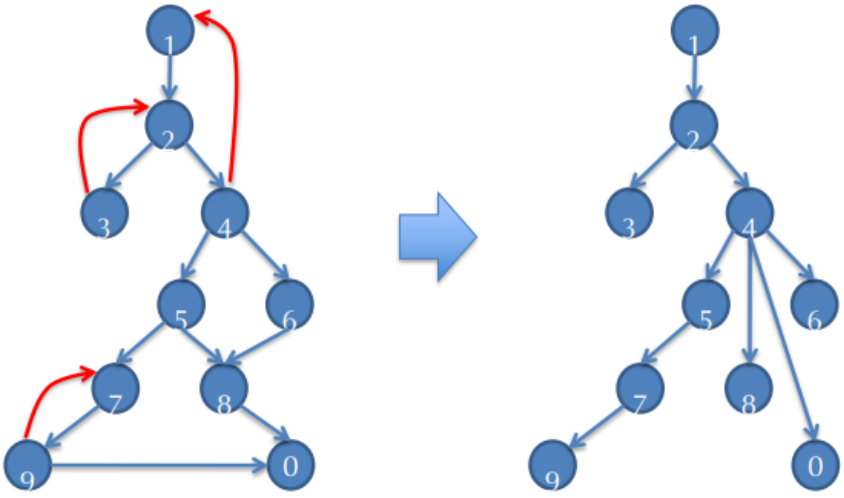
- Nested Loops
    - A CFG may contain many loops, loops may contain other loops



- Control-flow Analysis
    - Goal: Identify loops & nesting structure in a CFG
    - Control flow analysis is based on the idea of **dominators**
    - A **dominates** B: if the only way to reach B from start node is via A
    - An edge in the CFG is a back edge if its target dominates the source
    - A loop contains $\geq$ 1 back edge



- Dominator Trees
  - Dom is **transitive**: A dom B & B dom C $\rightarrow$ A dom C
  - Dom is **anti-symmetric**: A dom B & B dom A $\rightarrow$ A = B
  - Every flow graph has a **dominator tree**
    - The Hasse diagram of the dominates relation



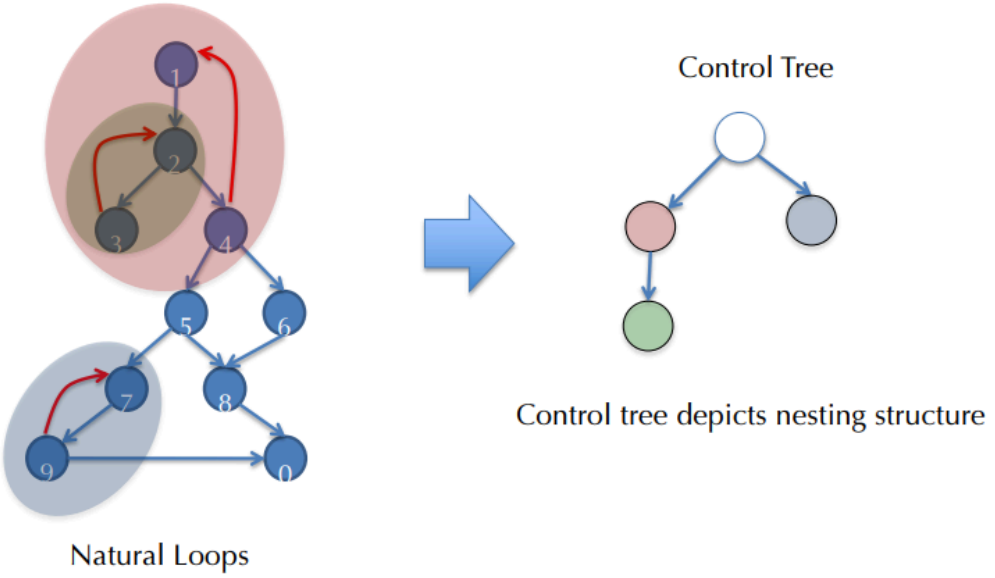
- Dominator Dataflow Analysis
  - We can define $\texttt{Dom}[n]$ as a forward dataflow analysis
    - $\texttt{Dom}[n]$: all the nodes that dominate n
    - Using the framework we saw earlier: $\texttt{Dom}[n] = \texttt{out}[n]$ where
    - B is dominated by A if A dominates all B's predecessors

      $\texttt{in}[n] := \bigcap_{n' \in \texttt{pred}[n]} \texttt{out}[n']$
    - Every node dominates itself

      $\texttt{out}[n] := \texttt{in}[n] \cup \{n\}$
    - $\texttt{out}[entry] = \{entry\}$, or $\texttt{in}[entry] = \varnothing$

        - $\texttt{out}[n] = \{\text{all nodes}\}, n \neq \textit{entry}$
    - Formally, $\mathcal{L}$ =set of nodes ordered by $\subseteq$
        - $T = \{\text{all nodes}\}$
        - $F_n(x) = x \cup \{n\}$
        - *JOIN* is $\cap$
    - Easy to show
        - F_n is monotonic
        - F_n distributes over meet
    - So, the algorithm terminates and is MOP
- Improving the Algorithm
    - $\texttt{Dom}[b]$: those nodes along the path in dominator tree from root to b (lots of sharing among the nodes)
    - More efficient to represent Dom sets by storing dominator tree
        - $\texttt{doms}[b]$ = immediate dominator of b
    - To compute $\texttt{Dom}[b]$, walk through $\texttt{doms}[b]$
    - Need to efficiently compute $\texttt{Dom}[a] \cap \texttt{Dom}[b]$
        - Traverse up tree and look for least common ancestor
- Completing Control-flow analysis
    - Dominator analysis identifies back edges
        - Edge $n \to h$ where h dominates n
    - Each back edge has a natural loop
        - h is the header
        - All nodes dominated by h that also reach n without going through h
    - For each back edge $n \to h$, find the natural loop

      $\{n' \mid h \text{ dom } n' \;\&\; n \text{ is reachable from } n' \text{ in } G \setminus \{h\} \cup \{h\}$
    - Two loops may share the same header (merge them)
    - Nesting structure of loops is determined by set inclusion (can be used to build the control tree)
- Example natural loops



- Uses of Control-flow information
    - Loop nesting depth plays an important role in optimization heuristics (deeply nested loops pay off the most for optimization)
    - Need to know loop headers and back edges for doing
        - loop invariant code motion
        - loop unrolling
    - Dominance information also plays a role in converting to SSA form
        - used internally by LLVM to do register allocation

- Revisiting SSA
    - Single Static Assignment
    - Alloca vs. `%uid`

Current compilation strategy

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```
%x = alloca i64
%y = alloca i64
store i64* %x, 3
store i64* %y, 0
%x1 = load %i64* %x
%tmp1 = add i64 %x1, 1
store i64* %x, %tmp1
%x2 = load %i64* %x
%tmp2 = add i64 %x2, 2
store i64* %y, %tmp2
```

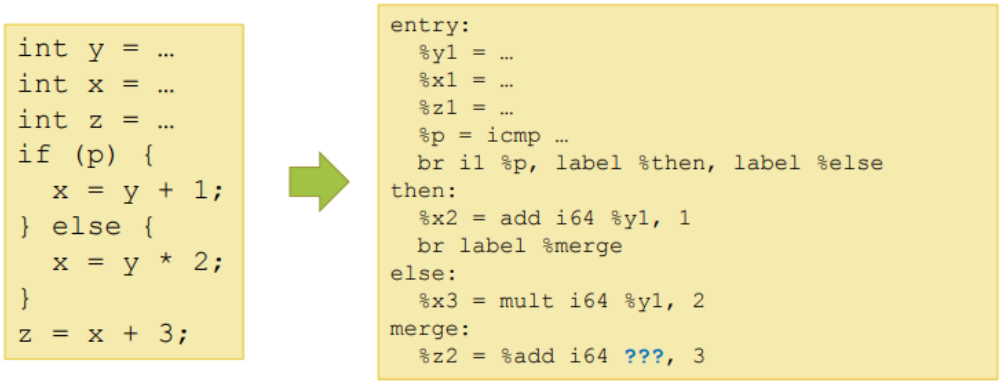
Directly map source variables into `%uids`?

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```
int x1 = 3;
int y1 = 0;
x2 = x1 + 1;
y2 = x2 + 2;
```

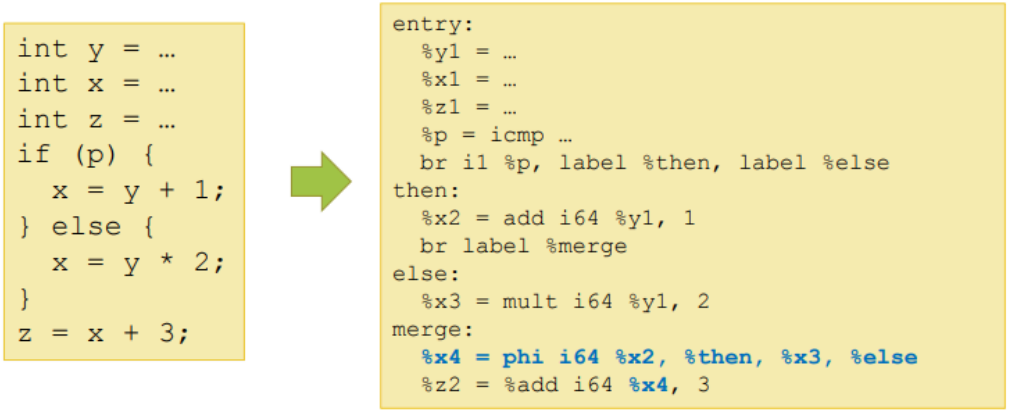
```
%x1 = add i64 3, 0
%y1 = add i64 0, 0
%x2 = add i64 %x1, 1
%y2 = add i64 %x2, 2
```

    - What about `if-then-else`



- Phi functions (ϕ functions)
    - fictitious operator, used only for analysis (implemented by **mov** at x86 level)
    - Chooses versions of a variable by the path how control enters phi node

%uid = phi <ty> v_1, <label$_1$>, … , v_n, <label$_n$>



- Phi nodes and loops
    - Importantly, **%uids** on RHS of phi nodes can be defined "later" in CFG
        - Meaning that **%uids** can hold values "around a loop"
        - Scope of **%uids** is defined by **dominance**

```
entry:
  %y1 = …
  %x1 = …
  br label %body

body:
  %x2 = phi i64 %x1, %entry, %x3, %body
  %x3 = add i64 %x2, %y1
  %p = icmp slt i64, %x3, 10
  br i1 %p, label %body, label %after

after:
  …
```

- Alloca Promotion
  - Not all source variables can be allocated to registers
    - if the address of the variable is taken

```
entry:
  %x = alloca i64          // %x cannot be promoted
  %y = call malloc(i64 8)
  %ptr = bitcast i8* %y to i64**
  store i64** %ptr, %x     // store the pointer into the heap
```

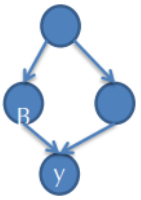
    - if the address of the variable "escapes" (by being passed to a function)

```
entry:
  %x = alloca i64        // %x cannot be promoted
  %y = call foo(i64* %x) // foo may store the pointer into the heap
```

  - An alloca inst. is **promotable** if neither of the above conditions holds
  - Most local variables declared in source programs are promotable
    - meaning they can be register allocated
- Converting to SSA: Overview
  - Start with ordinary CFG that uses allocas (identify "promotable" allocas)
  - Compute dominator tree information
  - Calculate def/use information for each such allocated variable
  - Insert ϕ functions for each variable at necessary "join points"
  - Replace loads/stores to alloc'ed variables with freshly-generated `%uids`
  - Eliminate the now unneeded load/store/alloca instructions
- Where to place ϕ functions

- need to calculate the "Dominance Frontier"
- Node A strictly dominates B if A dom B & $A \neq B$
- The dominance frontier of a node B is the set of all CFG nodes y such that B dominates a predecessor of y, but does not strictly dominate y

Intuitively: starting at B, there is a path to y, but there is another route to y that does not go through B

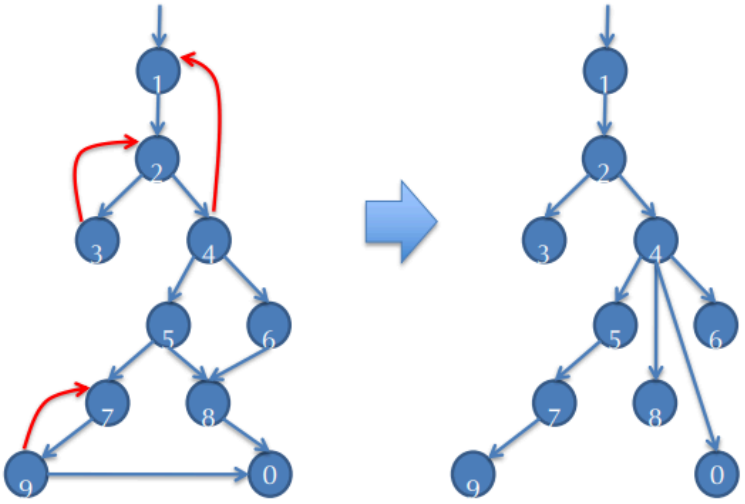


- Write $DF[n]$ for the dominance frontier of node n

- Example: dominance frontiers

Example of a dominance frontier calculation results

DF[1] = {1}, DF[2] = {1,2}, DF[3] = {2}, DF[4] = {1}, DF[5] = {8,0}, DF[6] = {8}, DF[7] = {7,0}, DF[8] = {0}, DF[9] = {7,0}, DF[0] = {}



- Algorithm for Computing $DF[n]$
  - Assume `doms`$[n]$ stores the dominator tree
    - that is the immediate dominator of n in the tree
  - Adds each B to the DF sets to which it belongs

```
for all nodes B
    if #(pred[B]) ≥ 2                    // (just an optimization)
     for each p ∈ pred[B] {
          runner := p                    // start at the predecessor of B
          while (runner ≠ doms[B])       // walk up the tree adding B
               DF[runner] := DF[runner] ∪ {B}
               runner := doms[runner]
    }
```

- Insert ϕ at Join Points

Lift the DF[n] to a set of nodes N in the obvious way

$$DF[N] = \bigcup_{n \in N} DF[n]$$

Suppose **variable x** is defined at a set of nodes N

$DF_0[N] = DF[N]$

$DF_{i+1}[N] = DF[DF_i[N] \cup N]$

Let J[N] be the **least fixed point** of the sequence

$$DF_0[N] \subseteq DF_1[N] \subseteq DF_2[N] \subseteq DF_3[N] \subseteq \ldots$$

That is, $J[N] = DF_k[N]$ for some k such that $DF_k[N] = DF_{k+1}[N]$

- J[N] is called the "join points" for the set N

We insert ϕ functions for the variable x at each node in J[N]

- $x = \phi(x, x, \ldots, x)$; (one "x" argument for each predecessor of the node)
- In practice, J[N] is never directly computed, instead use a worklist algorithm that keeps adding nodes for $DF_k[N]$ until there are no changes, just as in the dataflow solver

Intuition

- If N is the set of places where x is modified, then DF[N] is the places where phi nodes need to be added, but those also "count" as modifications of x, so we need to insert the phi nodes to capture those modifications too

- Phi Placement alternative
  - Place phi nodes "maximally" (i.e. at every node with ≥ 2 predecessors)

If all values flowing into phi node are the same, then eliminate it

```
%x = phi   t %y, %pred1   t %y  %pred2  … t %y %predK
// code that uses %x
⇒
// code with %x replaced by %y
```

  - Interleave with other optimizations

- LLVM Phi Placement

This transformation is also sometimes called register promotion

- older versions of LLVM called it "mem2reg" memory to register promotion

LLVM combines it with **scalar replacement of aggregates** (SROA)

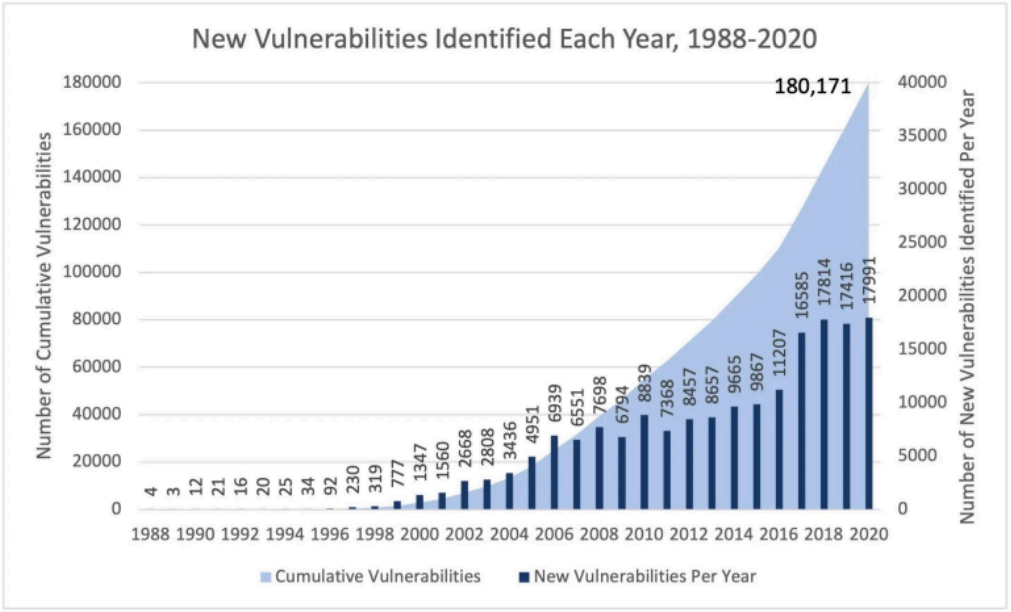
- i.e. transforming loads/stores of structured data into loads/stores on register-sized data

These algorithms are (one reason) why LLVM IR allows annotation of predecessor information in the .ll files

- Simplifies computing the DF

Nov. 29th - Lecture 22: Garbage Collection

- Automatic Memory Management (GC)
    - Plan
        - Why Automatic Memory Management?
        - Garbage collection
        - Three techniques: Mark and Sweep; Stop and Copy; Reference Counting
    - Why automatic memory management
        - Storage management has been a hard problem in modern programming
        - C/C++ programs have many storage bugs
            - forgetting to free unused memory
            - dereferencing a dangling pointer
            - overwriting parts of a data structure by accident
        - Storage bugs are hard to find
            - a bug can manifest far away in time & program text from its source
    - Software is getting buggier?



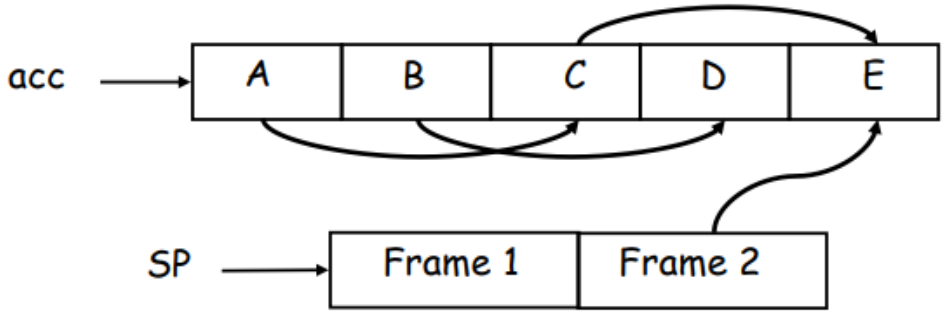
    - Type safety and Memory management
        - Some storage bugs can be prevented in a strongly typed language
            - we cannot overrun the array limits, dereference a null pointer

        - Can types prevent errors with manual memory allocation/deallocation?
            - some fancy type systems (linear types) were designed for this purpose (Rust)
            - ... but may complicate programming
        - It we want type safety, we typically must use AMM (GC)
- Automatic memory management
    - old problem: studied since the 1950s for LISP
    - several well-known techniques for performing completely AMM
    - for a (long) while, they were unpopular outside Lisp family of languages, just like type safety used to be unpopular
- The Basic Idea
    - When an object is created, unused space is automatically allocated (new objects are created by `malloc` or `new` in C/C++)
    - After a while there is no more unused space
    - Some space is occupied by objects that will never be used again
    - This space can be freed to be reused later
    - How to tell whether an object will "never be used again"?
        - heuristics to find **many** (not all) objects that will never be used again
    - Observation: a program can use only objects that it can find
        - `let x : A = new A in {x=y; ...}`
        - After `x = y` there is no way to access the newly allocated object
- Garbage
    - An object x is **reachable** iff
        - a register contains a pointer to x
        - another reachable object y contains a pointer to x
    - One can find all reachable objects by
        - starting from registers, and
        - following all the pointers
    - Unreachable objects can never be referred by the program
        - the objects are called **garbage**
- Reachability is an approximation

- Consider the program

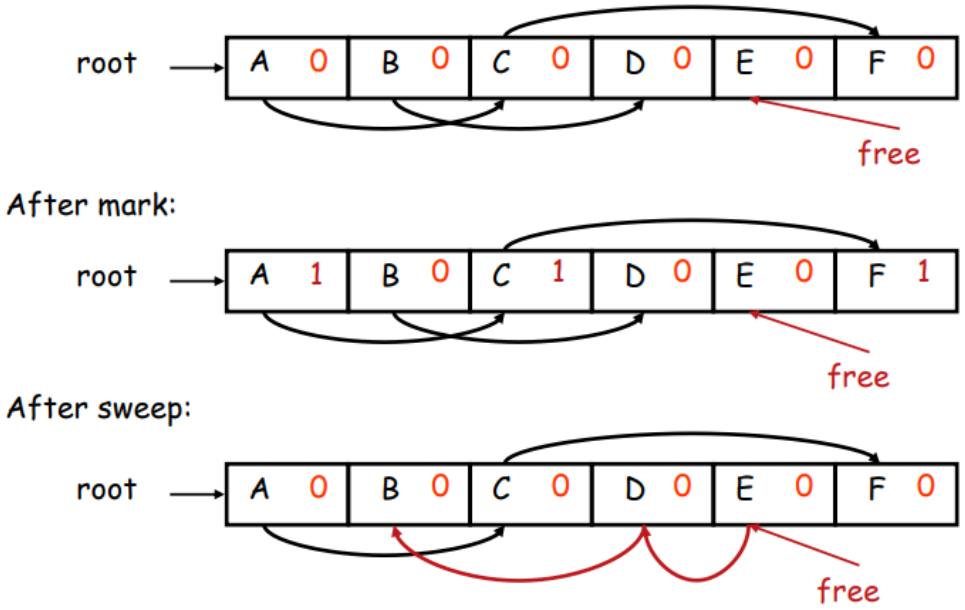
```
x = new A
y = new B
x = y
if alwaysTrue() then
    x = new A
else
    x.foo()
```

- After `x = y` (assuming `y` becomes dead there)
    - the object A is not reachable anymore
    - the object B is reachable (through `x`)
    - Thus, B is not garbage and is not collected
    - But, object B is never going to be used
- Tracing Reachable Values
    - Assume the only register is the accumulator
        - it points to an object, and
        - this object may point to other objects
    - The stack is more complex
        - each stack frame contains pointers (method parameters)
        - each stack frame also contains non-pointers (return address)
        - if we know the layout of the frame, we can find the pointers in it
- A simple example



    - We start tracing from acc and stack
        - the are called the roots
    - Note that B and D are not reachable from acc or the stack

        - Thus, we can reuse their storage
- Elements of GC
    - Every GC scheme has the following steps
        - allocate space as needed for new objects
        - When space runs out
            - compute what objects might be used again (generally by tracing objects reachable from a set of "root" registers)
            - Free the space used by objects not found in (a)
    - Some strategies perform GC before space actually runs out
- Mark and Sweep (McCarthy, 1960)
    - When memory runs out, GC executes two phases
        - mark: traces reachable objects
        - sweep: collects garbage objects
    - Every object has an extra bit: **mark bit**
        - reserved for memory management
        - initially the mark bit is 0
        - set to 1 for the reachable objects in the mark phase
    - Example



    - The Mark phase

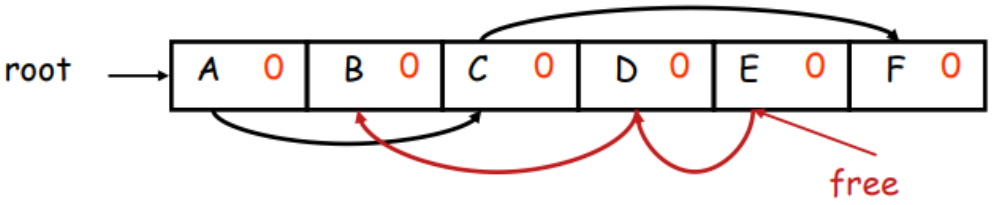
```
let todo = { all roots }
while todo ≠ ∅ do
    pick v ∈ todo
    todo ← todo \ { v }
    if mark(v) = 0 then      (* v is unmarked yet *)
       mark(v) ← 1
       let v1,...,vn be the pointers contained in v
       todo ← todo ∪ {v1,...,vn}
    fi
od
```

- The Sweep phase
    - The sweep phase scans the heap for objects with mark bit 0
        - these objects have not been visited in the mark phase
        - they are garbage
    - any such object is added to the free list
    - the objects with a mark bit 1 have their mark bit reset to 0

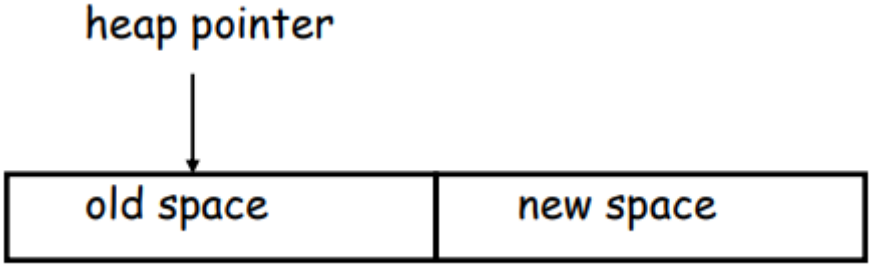
```
/* sizeof(p) is the size of block starting at p */
p ← bottom of heap
while p < top of heap do
  if mark(p) = 1 then
     mark(p) ← 0
  else
     add block p...(p+sizeof(p)-1) to freelist
  fi
  p ← p + sizeof(p)
od
```

- Details
    - While conceptually simple, there are some tricky details
    - A serious problem with the mark phase
        - it is invoked when we are out of space
        - yet it needs space to construct the `todo` list
        - size of the `todo` list is unbounded, so cannot reserve space a priori

        - The todo list is used as auxiliary data structure to perform reachability
        - There is a trick to allow the auxiliary data to be stored in the objects
            - pointer reversal: when a pointer is followed, reverse it to point to its parent
            - by Deutsch-Schorr-Waite (DSW)
        - Similarly, the free list is stored in the free objects themselves
- Pointer Reversal
- Mark and Sweep: Evaluation
    - Space for a new object is allocated from the new list
        - A block large enough is picked
        - an area of the necessary size is allocated from it
        - the left-over is put back in the free list

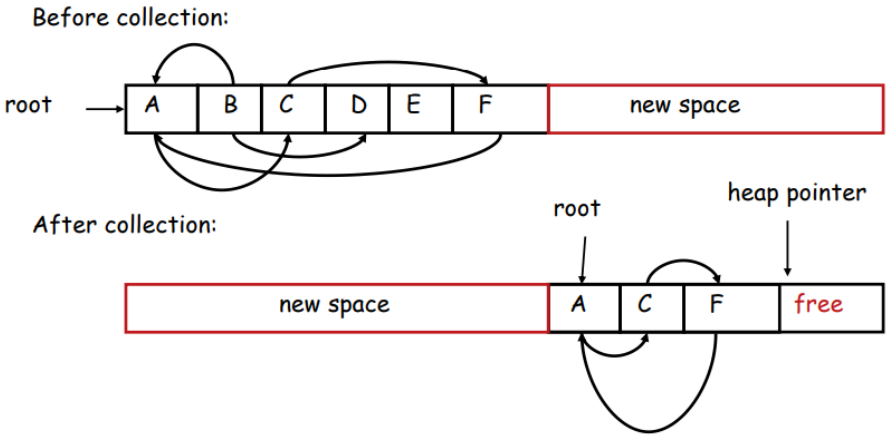


    - Mark and sweep can **fragment the memory**
    - Advantages: objects are not moved during GC
        - no need to update the pointers to objects
        - works for languages like C/C++
- Stop and Copy
    - Memory is organized into 2 areas
        - old space: used for allocation
        - new space: used as a reserve for GC

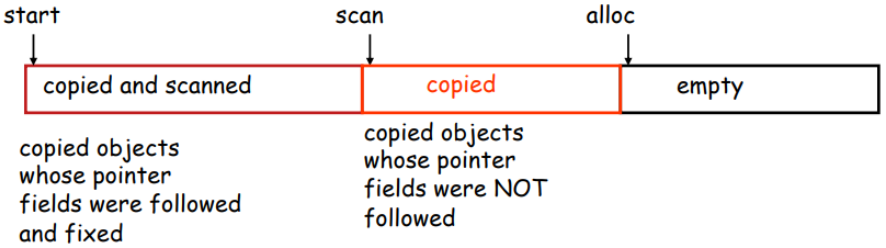


    - The heap pointer points to the next free word in old space
        - allocation just advances the heap pointer

- Stop and Copy GC
    - Starts when the old space is full
    - Copies all reachable objects from old space into new space
        - garbage is left behind
        - after copy phase, new space uses less space than old space before GC
    - After the copy
        - the roles of old and new spaces are reversed, and
        - the program resumes
    - Example



- Implementation of Stop and Copy
    - Need to find all reachable objects, as for mark and sweep
    - As we find a reachable object, copy it into the new space
        - and we have to fix all pointers pointing to it
    - As we copy an object
        - store in the old copy a **forwarding pointer** to the new copy
        - any object reached later with a forwarding pointer was already copied
    - still the issue of how to implement the traversal w/o using extra space
    - Partition the **new space** in 3 contiguous regions



- Stop and Copy Algorithm

```
while scan <> alloc do
    let O be the object at scan pointer
    for each pointer p contained in O do
        find O' that p points to
        if O' is without a forwarding pointer
            copy O' to new space (update alloc pointer)
            set 1st word of old O' to point to the new copy
            change p to point to the new copy of O'
        else
            set p in O equal to the forwarding pointer
        fi
    end for
    increment scan pointer to the next object
od
```

- Stop and Copy Details
    - Like mark & sweep, we must tell how large an object is when we scan it, and we must also know where are the pointers inside the object
    - We must also copy any objects pointed to by the stack, and update pointers in the stack (can be expensive operations)
- Stop and Copy: Evaluation
    - Stop and copy is generally believed to be the fastest GC technique
    - Allocation is very cheap, just increment the heap pointer
    - Collection is relatively cheap, especially if there is a lot of garbage, only touch reachable objects
    - but some languages do not allow copying (C/C++)
- Why doesn't C allow copying?
    - GC relies on being able to find all reachable objects, and it needs to find all pointers in an object
    - In C/C++, it's impossible to identify the contents of objects in memory
        - e.g. how to tell whether a sequence of two memory words is a list cell (data + next fields) or binary tree node (left + right fields)
        - we cannot tell where all the pointers are
- Conservative GC
    - It's okay/safe to be conservative

        - if a memory word looks like a pointer, it is considered a pointer (it must be aligned, it must point to a valid address in the data segment)
        - all such pointers are followed → we overestimate the reachable objects
    - But, we still cannot move objects as we cannot update pointers to them
        - what if what we thought to be a pointer is actually a number?
- Reference Counting (Collins, 1960)
    - Rather than wait for memory to run out, try to collect an object when there are no more pointers to it
    - Store in each object the number of pointers to that object (this is the reference count)
    - Each assignment operation has to manipulate the reference count
- Implementation of RC
    - `new` returns an object with a reference count of 1
    - if `x` points to an object, let `rc(x)` refer to the object's reference count
    - every assignment `x:=y` must be changed

```
rc(y) ← rc(y) + 1
rc(x) ← rc(x) - 1
if(rc(x) == 0) then mark x as free
x := y
```

- Evaluation of RC
    - Advantages
        - Easy to implement
        - Collects garbage incrementally without large pauses in the execution
    - Disadvantages
        - Manipulating reference counts at each assignment is very slow
        - Cannot collect circular structures
- GC: Evaluation

- Automatic memory management avoids some serious storage bugs
- But it takes away control from the programmer
    - layout of data in memory; when is memory deallocated
- Most GC implementations stop the execution during collection
    - not acceptable in real-time applications
- GC is going to be around for a while
- Advanced GC Algorithms
    - Concurrent: allow the program to run while collection is happening (concurrent mark & sweep)
    - Generational: do not scan long-lived objects at every collection (JVM)
    - Parallel: several collectors working in parallel

Dec. 2nd - Exercise 6: Optimizations

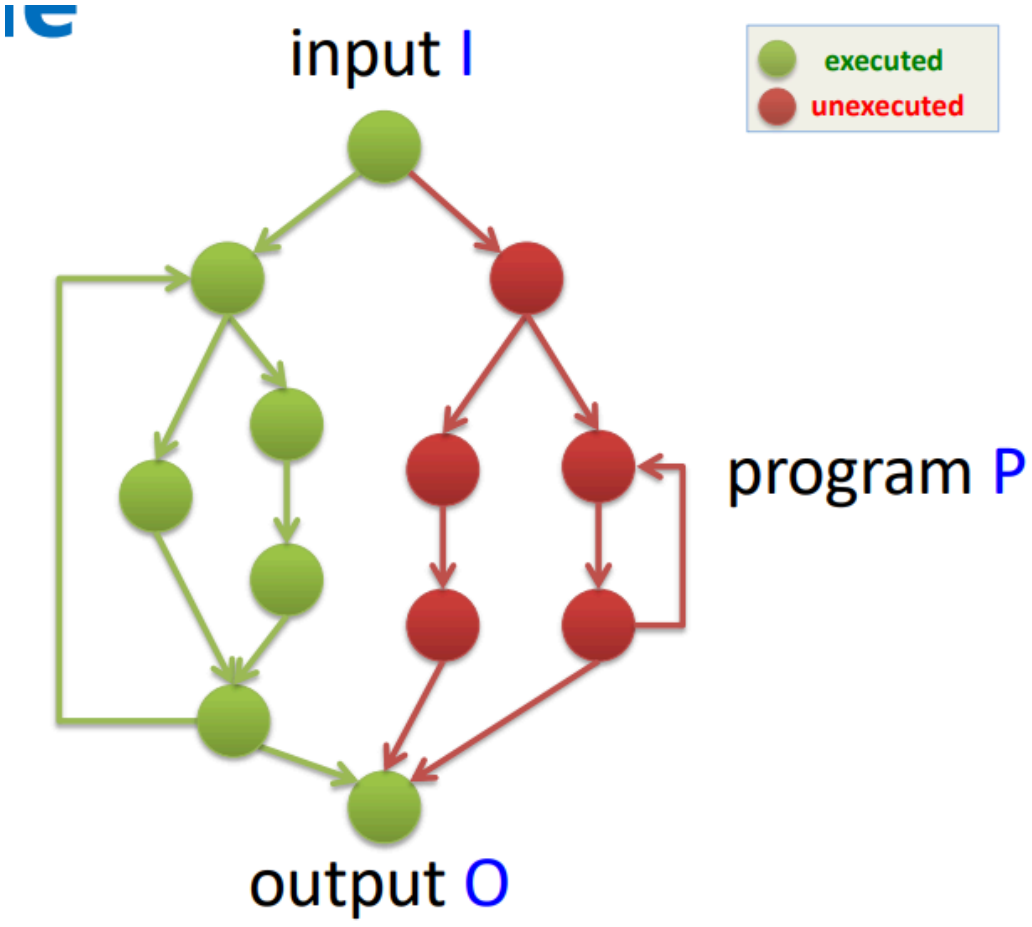
- Optimization
    - Constant Folding
    - Algebraic Simplification
    - Strength Reduction
    - Constant Propagation
    - Copy Propagation
    - Dead/Unreachable Code Elimination
    - Inlining
    - Common Subexpression Elimination
    - Loop Invariant Code Motion, Loop Unrolling

Dec. 4th - Lecture 23: Compiler Validation

- Compilers
    - Developers' belief: they are **faithful** translators
- Reflect on this belief
    - How trustworthy are compilers? How do they impact security & reliability?

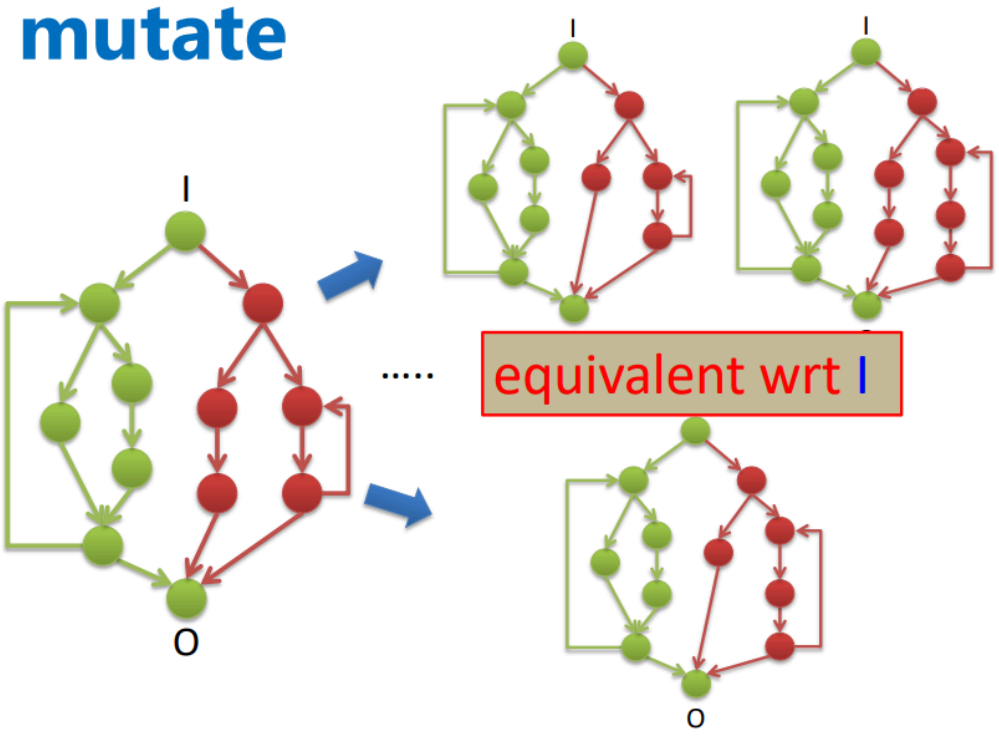
    - e.g. GCC 4.9 mis-compiled the Linux kernel.
- Type #1 bugs
    - LLVM bug 14972: valid code with no undefined behaviors, but `-O1` mis-compiles ("aborted: core dumped")
- Type #2 bugs
    - gcc "bug" 8536: `memset(Password, 0, sizeof(Password))`
    - This line is used to secure sensitive information by wiping the password from memory
    - Compiler removed this line for optimization purpose
- Type #3 bugs
    -
```
if (buf + len >= buf_end)
    return;     // len too large

if (buf + len < buf)
    return;     // overflow, buf+len wrapped around
```
    - Debatable
        - pointer overflow is **undefined behavior**
            - compiler assumes code has no undefined behavior
        - but this is a security threat
            - that is, using a large `len` to trigger **buffer overflow**
- Type #1 bugs: EMI Testing (find thousands of bugs in GCC/LLVM)
    - Csmith: using random test-case generation as source programs as input to C compilers, which yielded 325 bugs in total
    - Challenges
        - Generation: how to generate different, equivalent tests
        - Validation: how to check tests are indeed equivalent
    - Classical equivalence: $P = Q \iff \forall i, P(i) = Q(i)$
    - Equivalence Modulo Inputs: $P =^i Q \iff P(i) = Q(i)$
        - exploits close interplay between dynamic program execution on some input i and static compilation for all input
    - Profile



- The nodes are statements in program P

- Mutate



  - We do nothing on the executed statements, but we remove the unexecuted statements
  - If the compiler works correctly, removing unexecuted statements should affect the outcome of compilation

- Revisit LLVM bug 14972

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y,
  struct tiny z, long l) {
    if (x.c != 10) abort();
    if (x.d != 20) abort();
    if (x.e != 30) abort();
    if (y.c != 11) abort();
    if (y.d != 21) abort();
    if (y.e != 31) abort();
    if (z.c != 12) abort();
    if (z.d != 22) abort();
    if (z.e != 32) abort();
    if (l != 123) abort();
}
main() {
    struct tiny x[3];
    x[0].c = 10;
    x[1].c = 11;
    x[2].c = 12;
    x[0].d = 20;
    x[1].d = 21;
    x[2].d = 22;
    x[0].e = 30;
    x[1].e = 31;
    x[2].e = 32;
    f(3, x[0], x[1], x[2], (long)123);
    exit(0);
}
```


```
$ clang -m32 -O0 test.c ; ./a.out
$ clang -m32 -O1 test.c ; ./a.out
```

```c
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y,
  struct tiny z, long l) {
    if (x.c != 10) /* deleted */;
    if (x.d != 20) abort();
    if (x.e != 30) /* deleted */;
    if (y.c != 11) abort();
    if (y.d != 21) abort();
    if (y.e != 31) /* deleted */;
    if (z.c != 12) abort();
    if (z.d != 22) /* deleted */;
    if (z.e != 32) abort();
    if (l != 123) /* deleted */;
}
main() {
    struct tiny x[3];
    x[0].c = 10;
    x[1].c = 11;
    x[2].c = 12;
    x[0].d = 20;
    x[1].d = 21;
    x[2].d = 22;
    x[0].e = 30;
    x[1].e = 31;
    x[2].e = 32;
    f(3, x[0], x[1], x[2], (long)123);
    exit(0);
}
```

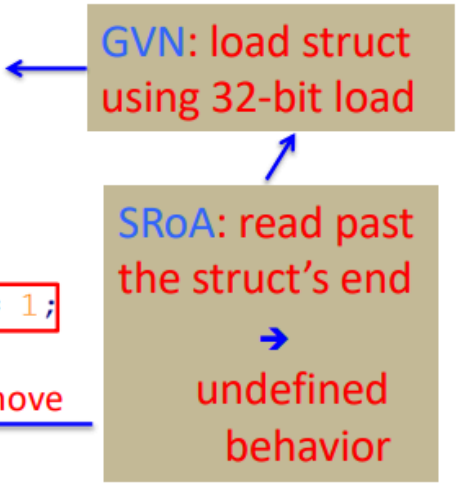
```
$ clang -m32 -O0 test.c ; ./a.out
$ clang -m32 -O1 test.c ; ./a.out
Aborted (core dumped)
```

- Bug autopsy

```c
struct tiny { char c; char d; char e; };

void foo(struct tiny x) {
    if (x.c != 1) abort();
    if (x.e != 1) abort();
}

int main() {
    struct tiny s;
    s.c = 1; s.d = 1; s.e = 1;
    foo(s);
    return 0;
}
```




```
$ clang -m32 -O0 test.c ; ./a.out
$ clang -m32 -O1 test.c ; ./a.out
Aborted (core dumped)
```

  - GVN (global variable numbering)
  - `struct tiny` has 3 `char` fields which are 24-bits long, but GVN reads 32-bits to load, this is compiler's undefined behavior

- When the compiler decides this is undefined behavior, it simply removes the line of value assignments

- Why does removing the `abort()` trigger the bug?

- Inlining optimization has a threshold, so when certain code is removed, the function body has reduced

- This thus triggers inlining which led to the bug

- gcc bug 58731

```
int a, b, c, d, e;
int main() {
    for (b = 4; b > -30; b--)
        for (; c;)
            for (;;) {
                e = a > 2147483647 - b;
                if (d) break;
            }
    return 0;
}
```

```
$ gcc -O0 test.c ; ./a.out
$ gcc -O3 test.c ; ./a.out
^C
```

- Look at `2147483647 - b`, it is a loop invariant, so optimization hoisted this segment as a variable

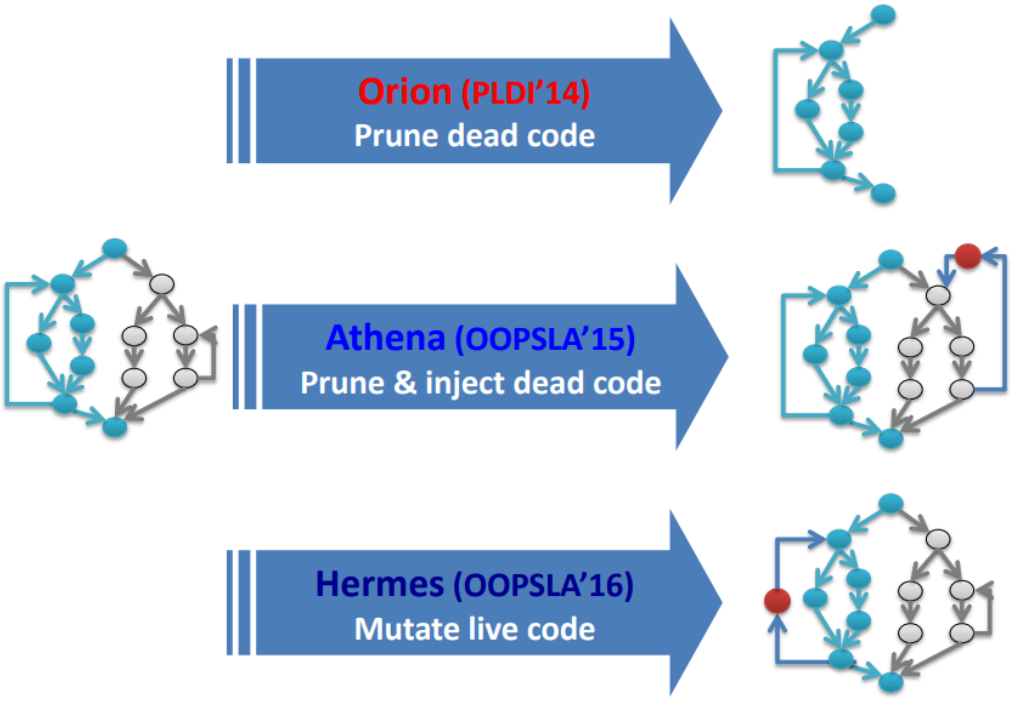
- But this hoist can cause an integer overflow

```
int a, b, c, d, e;
int main() {
    for (b = 4; b > -30; b--)
        int f = 2147483647 - b;
        for (; c;)
            for (;;) {
                e = a > f;
                if (d) break;
            }
    return 0;
}
```

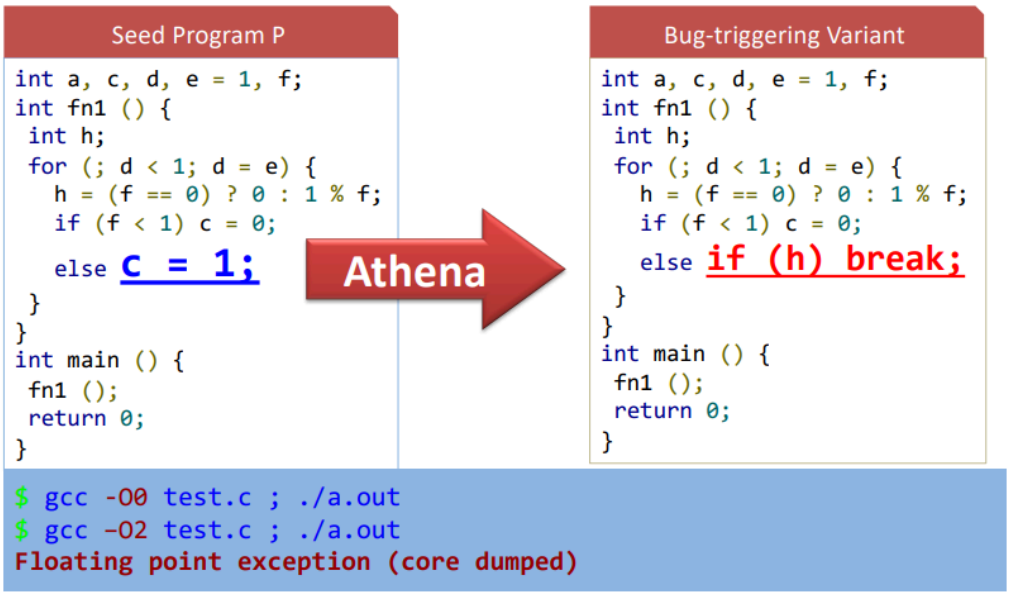
integer overflow

```
$ gcc -O0 test.c ; ./a.out
$ gcc -O3 test.c ; ./a.out
^C
```

- orion
  - First and simplest EMI realization
  - Targeting C compilers: randomly prune unexecuted code
- Bug importance
  - most bugs have already been fixed
  - many were critical, release-blocking
  - some affected real-world projects
- Different methods

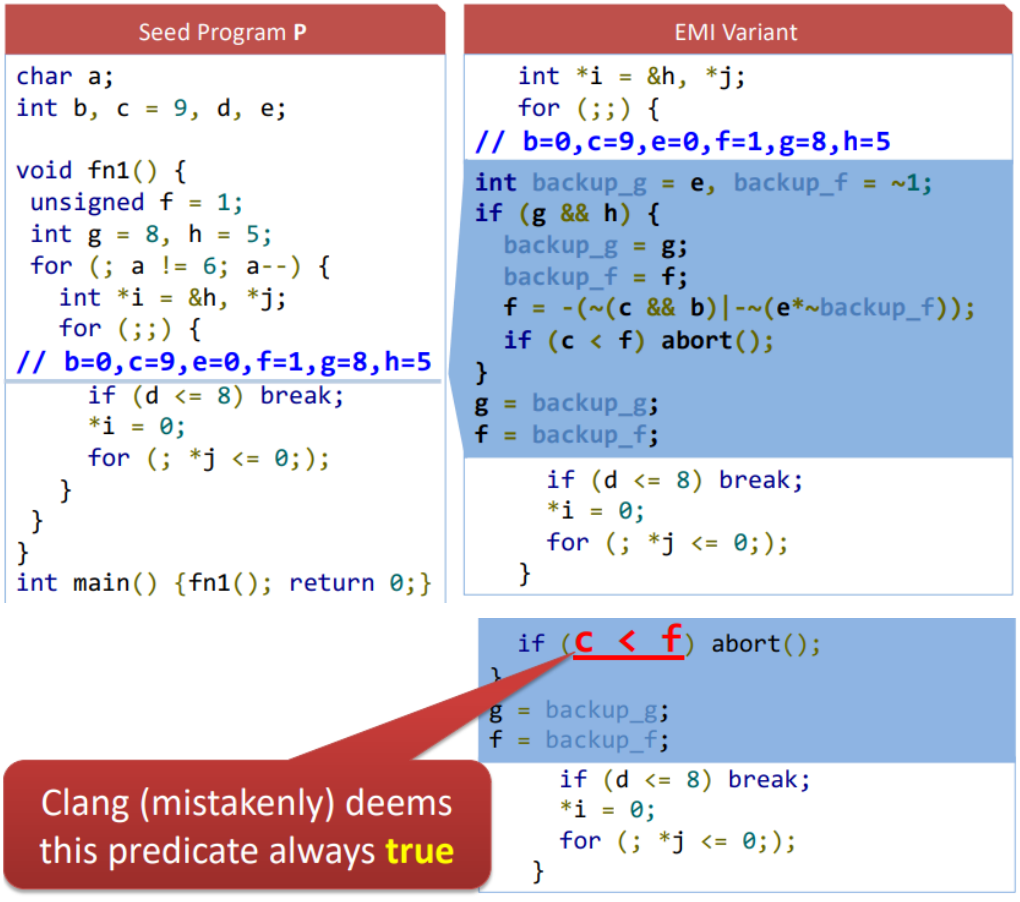


- Athena: gcc bug 61383



- Hermes
  - Profile and record variable values
  - Synthesize code snippets
    - ensure no undefined behavior
    - ensure EMI property locally
      - maintain same program state at entry & exit

- Hermese: LLVM 26266



- How about verification: Skeletal Program Enumeration (SPE)
- Program Enumeration

- Vision: bounded compiler verification
- Goal: Program enumeration
    - exhaustive (all small test programs)
    - practical
- Idea
    - Context-free grammar to token sequences
    - token sequences to programs (SPE)

- SPE

```
a := 10;                □ := 10;              b := 10;              a := 10;
b := 1;                 □ := 1;               a := 1;               b := 1;
while(a) do             while(□) do           while(b) do           while(b) do
  a := a − b;             □ := □ − □;           b := b − a;           b := a − b;
```

(a) Program P (b) Skeleton $\mathbb{P}$ (c) Program P_1 (d) Program P_2

- Use combinatorics to enumerate all the possible compilable programs
- Examples: wrong code, gcc crash, clang crash

- General, Effective
    - Hide all identifiers and enumerate following certain combinatorics
    - Simple to apply, no need to reduce

- How about `CompCert`

```c
// test.c
int main ()
{
   int a[2] = 0;
   return 0;
}
```

```c
// test.light.c
int main (void)
{
   int a[2];
   *(a+0) = 0;
   *(a+1) = 0;
   return 0;
}
```

- The codes on the left should not be accepted, but CompCert transforms it to the code on the right

```c
// test.c
#include <stdio.h>
int main(){
  int i = '\214';
  printf("%d\n", i);
  return 0;
}
```

```
$ ccomp-2.4 test.c; ./a.out
$ -116
$ ccomp-2.0 test.c; ./a.out
$ 140
```

```c
// test.c
volatile long long a;
unsigned b;
int main () {
  a = b;
  return 0;
}
```

```
$ ccomp-2.6 test.c
Fatal error: exception File "ia32/Asmexpand.ml", line 191, ...
...
$
```

- Guarantee: if can compile valid program successfully, then the executable behaves the same as the original program

```c
// test.c
#include <stdio.h>
int main () {
  int t = printf ("0\n");
  printf ("%d\n", t);
  return 0;
}
```

```
$ ccomp-2.6 -interp -quiet test.c
0
0
$ ccomp-2.7 -interp -quiet test.c
0
2
$
```

- `printf` returns a value in C, which the number of bytes of data being printed

```c
// test.c
int a, b, c, d, e, f, g;
void fn1 () {
 int h, i, j;
 if (g) {
    g = 1;
  L1: if (1) ;
  }
 short k = ~j;
 int l = 1 / (h & e & ~d + ((k & ~h) - ((1 | i) & (a | c)))), m = ~~j / (~h | d + a);
 j = l & m | h;
 if (j) ;
 j = k;
 int n = ~i | f, o = ~b - 1 / -n * ~i, p = f;
 goto L1;
}
int main () {
if (a) fn1 ();
 return 0;
}
```

```
$ ccomp-2.7 test.c
...
Fatal error: exception File "backend/Regalloc.ml", line 741, ...
...
$
```

        - Register allocation problem
- EMI
        - Can test compilers, analysis and transformation tools
        - Generates real-world tests
        - Requires no reference compilers
        - Have uncovered thousands of bugs in GCC and LLVM
        - Many were long latent and miscompilations
- Compilers may also miss optimizations (we want the compiler to generate **correct** and **good (efficient)** code)
- Solidifying Software Foundations
- How to build perfectly reliable software?

Dec. 6th - Lecture 24: Compiler Verification

Dec. 11th - Lecture 25: Multi-level Intermediate Representation (NextSilicon)

- Evolution of Compiler Pipelines
        - C/C++ → x86, directly
        - then, C/C++ → LLVM-IR → many architectures
        - then, many high-level languages → LLVM-IR → many architectures
        - High-level languages are having their own IRs these days (e.g. Swift, Rust, DSL)
        - Proposal of an upstream IR: ClangIR
- Redundancy in compiler IRs
        - Domain knowledge → Custom IRs → LLVM-IR
        - For Custom IRs, we need to implement
                - Parser, printer, pass-manager, dead-code elimination, peephole Opts
                - but this is costly,
                - we use auto-generated by MLIR
- A simple MLIR program

```
func.func @foo(%val : i64) -> i64 {
    %cst = arith.constant() { value = 42 : i64 } :
() -> i64
    %res = arith.addi %val, %cst : (i64, i64) -> i64
    func.return %res : i64
}
```

- Types: `i64`
- Operations: `func.func, arith.addi, func.return`
- Attributes (can be attached to operations): `{ value = 42 : i64 }`
- Custom parsing and printing can make IR more concise, for example, `arith.constant 42 : i64`

- Representing control-flow using basic blocks
  - Have tags for each block: `^bb, ^bb_true, ^bb_false, ^bb_end`, etc
  - Terminator operations link the blocks to form a control-flow graph
  - Basic blocks can also have arguments: `^bb_end(%v : i32)`
- Representing structured control-flow in MLIR
  - Can introduce `if-else` flow (through `scf` dialect), so that regions (e.g. an `else` block) can be attached to operations
  - `scf` is a more abstract dialect of the `cf` level, thus the name, **multi-level** IR
  - Unified representation let us mix dialects together
- MLIR compiler structure
  - Many dialects that form a tree
  - Entry from one dialect, and is able to continue along the tree to reach the leaf like LLVM-IR
- Defining a dialect in MLIR
  - Results = Operation & Dialect Name * Operand * Type
  - (University of Edinburgh) IRDL: An IR Definition Language for SSA compiler

```
Dialect Math {
    Type complex {
        Parameters (elementType: !IntegerType)
    }
    Operation mul {
        ConstraintVar (T: !complex<!IntegerType>)
        Operands (lhs: !T, rhs: !T)
        Results (res: !T)
    }
}
```

- Compilers in Industry (NextSilicon)
    - About NextSilicon
        - Hardware startup that mainly targets High-Performance Computing
        - Different fields where HPC is utilized: Energy, FinTech, Life Sciences, AI/ML & Advanced Data Ops, Graph Analytics, Manufacturing
    - Step 1: Identify the likely flow
        - Look for the part of code in a massive parallelization that takes large amount of run time
    - Step 2: Optimize the likely flow
        - A mill core is a software-defined processing core, optimized for a particular application by the runtime optimization process
        - Mill cores are optimized for throughput over latency, with high power efficiency
        - An optimized mill core can run hundreds or thousands data streams in parallel
    - Step 3: Replicate across the chip
        - Mill cores can be replicated hundreds of times for massively parallel processing
    - Why do we need compilers?
        - Language are not accelerator aware
        - Detecting and understanding parallelism
        - Hardware specific optimizations, e.g. block merging (if-conversion)
    - Why MLIR?
        - Extensibility

        - MLIR is extensible and allows reuse and collaboration
    - Abstractions
        - Can model parallel code, LLVM-IR can only handle sequential codes
        - Abstractions are explicit

```
scf.parallel (%iv) = (%lb) to (%ub)
step (%step) {
    // Parallel region (independent
iterations)
}
```

            Preservable during transformations
    - Modern infrastructure
        - Corrects LLVM design flaws: extensibility and parallelism
- How to get high-level dialects?
    - Classical MLIR inputs start at a high-level
    - HL Dialect → Lowerings → LLVM Dialect → Export → LLVM IR
    - Lifting Passes: LLVM IR → Import → LLVM Dialect → Lifting passes → Structured control flow
- Optimizations being worked on
    - Generic Mem2Reg and SROA
- Optimizations: SROA - Scalar Replacement of Aggregates

Dec. 13th - Lecture 26: GraalVM and Oracle DB (Oracle Lab)

- What is GraalVM?
    - High-performance optimizing Just-In-Time compiler
        - New machine code is generated during runtime
    - Ahead-of-Time (AOT) "Native Image" generator
    - Multi-language support (Truffle interpreter)
- One compiler, many configurations
    - Compiler configured for just-in-time compilation inside the Java HotSpot VM

        - Java HotSpot VM: JIT compilation
    - Compiler configured for ahead-of-time compilation
        - Native image generator: Points-to analysis, AOT compilation
    - Compiler configured for just-in-time compilation inside a Native Image
        - Native Image
- Compiler Details
    - Key Features of Graal Compiler
        - Written in Java: Clean code, Uses @Annotations instead of Macros for transformations
    - Aggressive high-level optimizations: example - partial escape analysis
    - Designed for speculative optimizations and deoptimization: Metadata for deoptimization is propagated through all optimization phases
        - Deoptimization: realize optimized generated code is not necessarily valid
    - Modular architecture: Compiler-VM separation
    - Foundation of over 60 publications at premier venues (PLDI, OOPSLA, CGO)
- Graph-based Intermediated Representation
    - Based on Sea of Nodes format (also used by HotSpot C2 Compiler)
    - SSA-form intermediate representation
    - Superposition of 3 Graphs
        - Control flow graph - red edges
        - Data flow graph - blue edges
        - ...
- Simple Optimizations
    - Important Peephole Optimizations
        - Constant folding, arithmetic optimizations, strength reduction
            - `CanonicalizerPhase`
            - Nodes implement the interface `Canonicalizeable`
            - Executed often in the compilation pipeline
            - Incremental canonicalizer only looks at new / changed nodes to save time
            - More complex peephole optimizations implement `Simplifiable` (Also run during CanonicalizerPhase, more rarely)

        - Example (ArrayLengthNode)
    - Global Value Numbering
        - Automatically done based on node equality
- Partial Escape Analysis
    - Escape Analysis
        - Problem: objects are always allocated
        - Objects "escape" the current scope if they leave the compiler's field of view (i.e. the graph)
        - Escaping means object is
            - passed as a parameter
            - written to field
            - return value
            - throw value
    - Leveraging Escape Analysis
        - What to do with this knowledge: object allocations are expensive, want to avoid it
        - Scalar replacement: replace the usage of object with a field
    - Degrees of Escape Analysis
        - Object does not escape compilation unit: object allocations are removed & fields are replaced with their scalarized values
        - Object does not escape thread: Lock Removal - If no other thread can see the object, we can remove locking
        - Object escapes: must allocate object in heap
    - Partial Escape Analysis
        - What if the code at runtime rarely requires the allocation (allocation only occurs in very few cases)?
            - Move the allocation to the part of code where it escapes
- Native Image
    - Build process
        - Input: All classes from application, libraries, and VM
            - Application, libraries, JDK, Substrate VM
        - Iterative analysis until fixed point is reached → Ahead-of-Time Compilation & Image Heap Writing
        - Output: Native executable

        - Image heap
            - Build time: 1. Compile sources, 2. load classes, 3. application initialization
            - Run time: 4. Run workload
            - Without GraalVM Native Image: 1, 2+3+4
            - With unoptimized Native Image: 1+2, 3+4
        - Benefits of AOT compilation
            - Fast startup
            - Low memory footprint
            - Security benefits
                - no code loading at runtime
- Points-to Analysis
    - Limitation: Closed-World Assumption
        - The points-to analysis needs to see all Java bytecode which may execute
            - We AOT compile all code
            - No interpretation option
        - No loading of new classes at run time
        - Dynamic parts of Java require configuration at build time
            - Reflection, JNI, Proxy, Resources
        - Changing dependencies requires rebuilding the image
- Native Image Optimization
    - Benefits of closed-world AOT compilation: predictable performance
        - No "deoptimization"
        - Larger code transformations at build time without worrying about deoptimization
        - Indirect method calls are simple and always constant time
    - Optimizations from Analysis Results
        - Remove fields never accessed
        - Reduce number of null checks
        - Inject more specific type information into graphs
            - Helps later with inlining
        - Remove unreachable code paths
        - Identify never-written values

- Profile-Guided Optimizations (PGO)
    - AOT compiled code cannot optimize itself at run time
        - No dynamic "hot spot" compilation
    - PGO requires relevant workloads at build time
    - Optimized code runs immediately at startup, no "warmup" curve
- Multilingual Engine
    - Oracle Database MLE: GraalVM in the Database
        - Started as an Oracle Labs research project some time in 2013
        - Allows to run and store JavaScript directly in Oracle Database
        - What makes MLE great?
            - no network latency, runs server-side in the database
            - modern programming language support
            - abundance of open-source third-party libraries
            - simpler integration into existing workflows
        - Why build MLE with GraalVM?
            - Support and interoperability for modern languages
            - MLE platform is built using Native Image (fast startup, interaction with native code, etc.)
    - Case study: Argument Conversion
        - How is vector converted from a SQL to a JS data-type
        - Downside: potentially large allocation + full copy for every function call
            - native pointer to buffer is passed from database to MLE
            - MLE allocates a JS Float32Array on the heap
            - Float vector elements copied into the array
        - Fact: for many small functions, the function arguments are the only allocations in a call
            - throughput bounded by GC
            - Allocation rate is high
        - Better idea: Float32Array directly references native buffer
        - Escape analysis

            - if won't escape: use the native pointer to data, reuse memory
                - no copy, no allocation
                - reuses previous native buffer for subsequent calls
                - must reason about lifetime
            - if will escape: allocate + copy
            - partial escape analysis does a great job at avoiding object allocations, but can't help here (unbounded vector size, does not have domain knowledge)
    - Copy Avoidance in MLE
        - a custom compiler phase
        - leverages existing partial escape analysis to get lifetime information on function arguments
        - Split the SQL → JS conversion into 2 components
            - The data buffer
            - A holder object on which lifetime can be probed
            - Invariant: holder object escapes iff data buffer (...?)
        - During JIT compilation: probe escape analysis and see if holder object escapes
        - During execution: if the holder object does not escape, use native memory directly
    - Handling Deoptimizations
        - Any buffer which avoided allocation needs to be promoted to an actual allocation
        - We do this lazily, after the function call
            - if no deoptimization happened at end of function call: release buffers, lifetime ended
            - if deoptimization happened: promote buffers

Dec. 16th - Exercise 7: Recap Garbage Collection & Control Flow Analysis

- Garbage Collection
    - Mark and Sweep
        - mark phase: traces reachable objects

        - sweep phase: collects garbage objects
        - extra bit for every object
        - concurrent implementation of Mark and Sweep for Go, older Java version
        - Why do we need pointer reversal? Traversing the reachability graph requires extra memory, but we do not have it.
    - Stop and Copy
        - Find all reachable objects as for Mark and Sweep
        - Copy reachable objects to new space
            - fix all pointers with new memory address
        - Does not work for C++: we cannot distinguish pointers and values as they can be cast to each other
    - Reference counting
        - On every assignment, the reference pointer increases by 1
        - Available in Rust (Arc)
        - Limitations
            - slow-ish since manipulation on every assignment
            - cannot collect circular orphans ($a \to b, b \to a$, but no object can reach a or b)
- Control Flow Analysis
    - Dominator Trees
        - Goal: identify loops and nesting structure in a CFG
        - Control flow analysis is based on the idea of **dominators**
            - A dominates B: if the only way to reach B from start node is via A
    - Definition of a loop
        - A loop is a set of nodes in the CFG with one distinguished entry: the header
        - each node reachable from header
        - the header reachable from each node
            - loop is a strongly connected component
        - no edges enter a loop except to a header
        - Exit node: node with exiting edges
    - Solution: ϕ functions

Dec. 20th - Final Lecture: Summary

- Final Exam
    - Everything up until (including) Garbage Collection + 6 HWs
- Why Compiler Design?
    - Different study goals
- Materials skipped
    - Concrete syntax/parsing
    - Source language features: exceptions, advanced type systems, type inference, concurrency
    - Intermediate Languages: design, bytecode, bytecode interpreters, just-in-time compilation (JIT)
    - Compilation: continuation-passing transformation, efficient representations, scalability
    - Optimization: scientific computing, cache, instruction selection/scheduling
    - Runtime support: advanced garbage collection algorithms
- What next?
    - Major relevant conferences: **PLDI**, POPL, OOPSLA, ICFP, **ASPLOS**, **CGO**, **CC**, ESOP
    - Technologies/Open-source projects
        - Yacc, lex, bison, flex
        - LLVM $\rightarrow$ MLIR
        - Java virtual machine, Microsoft's common language runtime (CLR)
        - WebAssembly
        - Different programming languages
- Applicable domains
    - General programming
        - In C/C++, better understanding of how the compiler works can help generate better code
        - ability to read assembly output from compiler
        - experience with functional programming can give different ways to think about how to solve a problem
    - Writing domain specific languages

        - Tools like lex/yacc (flex/bison, ocamllex/menhir) useful for little utilities
        - understanding abstract syntax and interpretation
    - Understanding hardware/software interface
        - Different devices have different instruction sets...
- Thesis Projects
    - Language Design and Implementation
    - (Futuristic) Programming methodologies, environments and tools
    - Program analysis, verification, and testing
    - Software (including emerging software) quality, reliability, and security
    - Education technologies (for compilers, programming, and CS in general)
- Reflections and Perspectives
    - Key mission of Computer Science?
        - (To help people turn their creative ideas into working software)
        - Building a "blackbox" that transforms such ideas into practical software
        - (Mission: Advance and Reimagine the Science and Practice of Software Construction)
    - Dimensions in Software
        - Reliability - How to build a perfectly reliable software?
            - Software crisis → Specification crisis: is the program doing what it is intended to do?
        - Performance - How to build that perfect compiler?
            - Source → Compiler (Complex heuristics) → Target
            - Understanding the gap between the current and the **optimal**: "What if we had **perfect X**?"
        - Productivity - How to build a perfect programming assistant? How to build a perfect programming language?
            - Capturing and reusing knowledge: ultimate Q&A system for code
            - Reducing the syntax vs. semantics gap (Visual semantic language: Algot)
    - Ultimate dreams
        - Software bug detector, optimal compiler, programming assistant, programming language