Introduction

- Compiler: Translator between programming lang.
- Source Code: Optimized for Humans:
  - Expressive: matches human idea
  - Redundant: to catch errors
  - Abstract: exact computation
- Low-Level Code: Optimized for Machine:
  - Reduction of: • Readability • Redundancy • ambiguity • Abstraction • Information
- Translation goals: • Correctness • Maintainable Code • Performant Code • Efficient Translation
- Source level expressiveness of task
- Structure:

Source Code as Char Stream
↓ Char Stream
Front End (machine dependent): Lexical Analysis
↓ Token Stream
Parsing
↓ Abstract Syntax Tree
Middle End (compile dependent): Intermediate Code Generation
↓ Intermediate Code
Back End (machine dependent): Code Generation
↓ Assembly Code

- Typical Stages:
  - Lexing → token stream
  - Parsing → abstract Syntax
  - Disambiguation → abstract syntax
  - Semantic analysis → annotated abstract syntax
  - Translation → Intermediate Code
  - Control-flow analysis → Control Flow graph
  - Data-flow analysis → Interference graph
  - Register allocation → assembly
  - Code Emission
- Optimisations may be done at any stage
-

X86 Lite

- Operands:
  - Imm: 64-bit literal signed integer w. prepend. $
  - Lbl: label representing some machine address
  - Reg: val. of 1 of 16 registers w. prepended %
  - Ind: Machine address: base: Reg (index: Reg, displacement: int32) ⇒ base + index*8 + disp.
    - used as loc: ind • used as val: mem[ind]
- Instructions: • only quadword • src before dest
  - movq src, dst • dest: reg • addr
  - src: reg • addr • Imm • lbl
- Arithmetic Instructions
  - negq dest • 2's complement negation
  - addq src, dest • dst = src + dest
  - subq src, dst • dest = dest - src
  - imulq src, reg • reg = reg * src • truncated to 128 bit
- Logic / Bit Manipulation Instructions
  - notq dest • logical negation • bitwise not
  - andq src, dst • dst = dst & src • bitwise and
  - orq src, dst • dst = dst | src • bitwise or
  - xorq src, dst • dst = dst xor src • bitwise xor
  - sarq amt, dst • dst = dst >> amt • arith. shift right
  - shlq amt, dst • dst = dst << amt • logic. shift left
  - shrq amt, dst • dst = dst >> amt • logic shift right
- Conditional Instructions
  - cmpq src2, src1 • sub src1 - src2 and set condition flags
  - setbCC dest • dest's lower byte = 1 if CC else 0
  - jCC src • rip = src if CC else fallthrough
    - Condition Flags
      - OF • overflow • SF • sign • ZF • zero
    - Condition Codes
      - e (equality) → ZF
      - ne (inequality) → (not ZF)
      - g (greater than) → (not ZF) and (SF = OF)
      - l (less than) → SF ≠ OF
      - ge (greate equal) → SF = OF
      - le (less equal) → SF ≠ OF or ZF
- Function Instructions
  - jmp src • rip = src • jump to location
  - callq src • push rip ; rip = src
  - retq • rip = pop • i.e. jump to addr which is on top of stack
- Memory Instructions
  - leaq ind, dst • dst = addr(ind)
  - pushq src • rsp = rsp - 8; Mem[rsp] = src
  - popq dst • dst = Mem[rsp]; rsp = rsp + 8
- Labeled Blocks: • Jump to targets • begin at main
- Registers: • RIP • RAX • RBX • RCX • RSI • RDI • RBP • RSP • R08 • R09 • R10 • R12 • R13 • R14 • R15
  - Arguments: • RDI • RSI • RDX • RCX • R08 • R09
  - Callee save: • RBP • RBX • R12 • R13 • R14 • R15
  - Return: • RAX • Instruction Ptr: • R%P

Intermediate Representation (IR)

- Hides details of the target architecture
- Allows machine independant code gen. and opt.
- Common representation for many sources / dests.
- Goal: Low-level code without loosing information for analysis and optimisation.
- Multiple IRs may be used in sequence
- High-level IR:
  - Abstract syntax & new node types not gen. by parser
  - Represents high-level language constructs
  - High-level optimisations based on program structure
  - Use full for semantic analysis like type checking
- Low-Level IR:
  - Machine dep. assembly code + extra pseudo-instructions
  - Source structure of program is lost
  - Low-level optimisations based on target architecture
- Mid-level IR:
  - Between abstract syntax and assembly
  - Have: • unstructured jumps • abstract regs • mem. locs.
  - Basic Block: • Sequence of instructions • always start at the first instr. • always end at last instr.
    - starts with label (entry point)
    - ends with control-flow instr. (link) / terminator)
    - Contains no (other) :: control-flow instr. • label
  - Control-Flow Graph (CFG):
    - Node: Basic Blocks
    - Edge: ‖ ¶ control-flow instr. jumps to next node

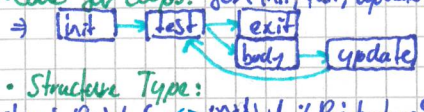
LLVM

- Single Static Assignment (SSA): Var. is assigned at most once
- Instruction: %L = OP TYPE OP1, OP2
  ⇒ %L = OP1 op OP2
- alloca: abstract storage on stack
  - alloca TYPE: returns ptr to it

```
%acc = alloca i64
store i64 341, i64* %acc
%x = load i64, i64* %acc
```

```
@s = global i32 42
declare void @use(i64)
define i64 @foo(i64 %a, i64* %b) {
  %sum = add nsw i64 %a, 42
  %cond = icmp sgt i64 %sum, 100
  br i1 %cond, label %then, label %else
then:
  call void @use(i64 %sum)
  ret i64 %sum
else:
  store i64 %sum, i64* %b
  ret i64 %sum
}
```

- Code for Loops: for(init, test, update){ body }



- Structure Type:

```
%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
}
```

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

- getelementptr (GEP):

```
struct R1 {
  int A;
  int B[10][20];
  int C;
}
```
⇔
```
%R1 = type {
  i32,
  [10 x [20 x i32]],
  i32
}
```

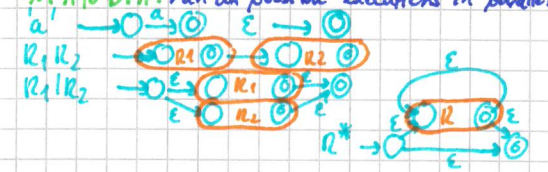
```
struct R2 {
  struct R1 X;
  int Y;
  struct R1 Z;
}
```
⇔
```
%R2 = type {
  %R1,
  i32,
  %R1
}
```

```
int* foo(struct R2* %s) {
  return &s[1].Z.B[5][13];
}
```

```
%a = getelementptr %R2* %s, i32 1, i32 2, i32 1, i32 5, i32 13
   = ADDR + s(%R2) + s(%R1) + s(i32) + s(i32)
     + 5 * 20 * s(i32) + 13 * s(i32)
```

Lexer

- Input: Char Stream Output: Token Stream
- Token: Datatype to represent indivisible chunks
- Regular Expression: Describe sets of strings
- Lexer generator: Rules consisting of regex with associated action
- Implementation: • Table based • DFA
- Finite Automata: Can represent any regex
- DFA: Accept input if consumed in accept state
- NFA: Accept input if ∃ way to reach accept state
- NFA to DFA: run all possible executions in parallel



Parser

- Input: Token Stream • Output: Abstract Syntax Tree
- Context-Free Grammars (CFG): Derive String by applying some rules to starting token
- Consists of: • Set of Terminals: elements w/o derivation rule • Set of Nonterminals: elem. w. derivation rules • Start Symbol: designated non-terminal • Set of Productions: LHS → RHS
- Accepts String: Iff ∃ derivation rule to get to this string
- Parse Tree: Represents derivation • Leaves: terminals • Inner Nodes: non-terminals • Easy to convert to AST
- Derivation: Two ways:
  - Leftmost: apply production to left-most non-terminal
  - Right-most: apply production to right-most non-terminal
  - Both produce the same parse tree
- Associativity: Ex. of a right-recursive (right associative) grammar | S ↦ E+S | E
E ↦ Num | (S)
- ambiguity: Grammar is left and right associative
- Fix: • Add non-terminals and allow recursion only on left or right • Higher-priority operators go further from the start symbol
- Ex: S ↦ S-S | S+S | (S) | num
⇒ $S_0 \mapsto S_0 + S_1 \mid S_1$, $S_1 \mapsto S_1 * S_2 \mid S_2$, $S_2 \mapsto$ num | (S_0)
- Unambiguous CFGs may still have multiple derivation • All have same AST

- Derivation: Two ways:
- Top-Down: Start from start symbol and go down • Need to guess which production to use • gives good parser Errors
- Bottom-Up: Start from final string
- Left-to-right parse, Left-most deriv, 1 lookahead (LL(1))
- Make Grammar LL(1): • Left-Factor Grammar: If ∃ common S prefix for each choice, then add a new non-terminal S' at decision pt: • Ex:

S ↦ E+S | E E ↦ num | (S)	⇒ S ↦ ES' S' ↦ ε | +S E ↦ num | (S)

- Eliminate left-recursion: Rewrite: $S \mapsto S\alpha_1 \mid \dots \mid S\alpha_n \mid \beta_1 \mid \dots \mid \beta_m \Rightarrow S \mapsto \beta_1 S' \mid \dots \mid \beta_m S'$
$S' \mapsto \alpha_1 S' \mid \dots \mid \alpha_n S' \mid \varepsilon$
- Ex:

S ↦ S+E | E E ↦ num | (S)	⇒ S ↦ ES' S' ↦ +ES' | ε E ↦ num | (S)

- Predictive Parsing Table:
- nonterminal x input token → production • Ex:

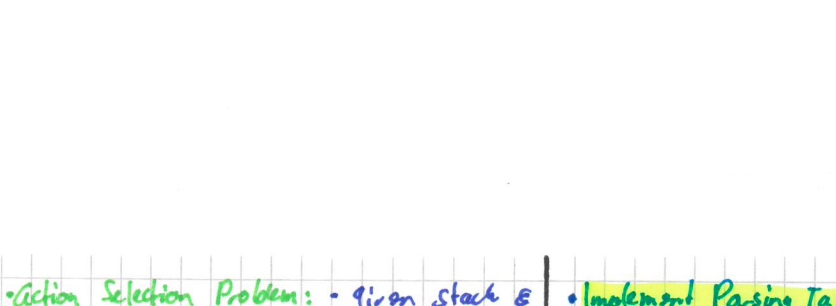
	num	+	(	)	$ (EOF)
T ↦ S$					
S ↦ ES'	T	↦S$		↦S$	
S' ↦ ε	S	↦ES'		↦ES'	
S' ↦ +S	S'		↦+S		↦ε ↦ε
E ↦ num | (S)	E	↦num		↦(S)	

- Construct Parser Table: Consider A → γ
- C1: Construct set of all input tokens that may appear first in strings that can be derived from γ ⇒ add A x token = ↦ γ ∀ such token
- C2: If γ can derive ε, then construct set of all input tokens that may follow the non-terminal A ⇒ add A x token = ↦ γ ∀ such token
- If ∃ two different productions for the same entry ⇒ grammar is not LL(1)
- First(T): Set of tokens that can be first when deriving T
- Follow(S): Set of tokens that can be first after S • If ∃ A → pB ⇒ Follow(A) ⊆ Follow(B)
- LR(k): • Left-to-right scanning • right-most derivation • k lookaheads
- more expressive than LL(k): can handle left and right recursive grammars
- Shift + Reduce: • Work bottom-up • Sequence of shift and reduce operations:
- Shift: move lookahead token to stack
- Reduce: replace symbols γ at top of stack w. nonterminal X s.t. X → γ is a production
- Parser State: • Stack: terminals and nonterminals • Input: String of terminals
- Ex. LR(0): very weak

- Action Selection Problem: • given stack δ and lookahead b, should parser:
  - Shift b onto stack (new stack is δ b) or
  - reduce production X → γ, assuming δ = α γ (new stack is α X)
  - Idea: Decide based on prefix α of stack plus lookahead
  - Goal: Know set of legal productions at any pt in time
- Implementation: Production with an extra "." in the RHS • Stuff before "." is already on stack • Stuff after "." is what might be seen next
- LR(0) Item
- Construct DFA: 1: Add new production S' ↦ S$ to grammar • 2: Start state is empty stack, so it contains S' ↦ .S$
- Closure of a State: • adds items for all productions whose LHS nonterminal occurs in an item in the state just after the "." (e.g. S in S ↦ .S$) • The added items have the "." located at the beginning • The newly added items may cause yet more items to be added to the state ⇒ iterate till fixpoint
- Ex:

S' ↦ S$ S ↦ (L) | id L ↦ S | L,S	Closure({S' ↦ .S$}) = { S' ↦ .S$, S ↦ .(L), S ↦ .id}

- Ex. DFA: • 3: Calculate closure of current state • 4: add transitions for shift "." for all new states calculate closures • 6: repeat ...

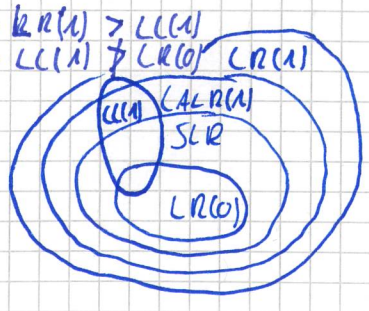


- Optimisations: Need to
- Usage: Run parse through DFA. Stack tells what production to use next: • Input in reduce state: Shift the next symbol and transition w.r.t. DFA • If in reduce state: X → γ with αγ, pop γ and push X
- Optimisation: Prevent rerunning whole DFA: • Store state with each symbol on stack, e.g. $_1(_2(_3L_5)_6$ • On reduction X ↦ γ, pop stack to reveal the state too, e.g. from $_1(_3(_2L_5)_6$ reduce S ↦ (L) to $_1(_3$ • Push the reduction symbol, e.g. get $_1(_3S$ • Take one step in DFA to find next state, e.g. $_1(_3S_7$

- Implement Parsing Table: • Represent DFA as table of shape: state ↦ (terminals + non-terminals) → state (terminals: action table; non-terminals: goto table)
- Entries of action table specify two kind of actions:
  - 1: Shift and go to state n
  - 2: Reduce using reduction X ↦ γ: • a) pop γ off the stack to reveal the state • b) Lookup X in the goto table and go to that state
- LR(0) only works if states with reduce actions have a single reduce action
- More complex grammars lead to:
  - shift/reduce conflicts: S ↦ (L). | L ↦ L,S
  - reduce/reduce conflicts: S ↦ L,S. | S ↦ ,S.
- Can often be resolved using 1 lookahead ⇒ LR(1)
- LR(1): • LR(1) State: Set of LR(1) items
- LR(1) Item: LR(0) item + a set of lookahead symbols
- LR(1) Closure: • Form set as in LR(0) case • whenever C ↦ .γ is added bc. A ↦ β.Cδ, L is already in the set, compute lookahead set M: • 1. M includes FIRST(δ) • 2. If δ is or can derive ε, then M also contains L • Ex:

1 start:	S' ↦ .S$, {$}	S' ↦ S$
2 add:	S ↦ .E+S , {$} S ↦ .E , {$}	FirstS($) = $ S ↦ E+S | E E ↦ num | (S)
3 add:	E ↦ .num , {+} E ↦ .(S) , {+}	∈ First(+S) = {+}
4 add:	E ↦ .num , {$} E ↦ .(S) , {$}	ε ∈ δ is ε ⇒ add L

- LL: handle left & right recursive
- LR: handle right recursive
- LALR: introduce R/R, but no S/R conflicts
- k LR(k) ⊇ LL(k)
- LL(k) ⊉ LR(0)

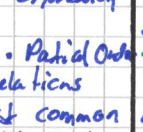


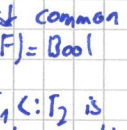
Untyped Lambda Calculus

- Substitution: $e[v/x]$ replace all free occurrences of x in e by v
- Free Variable: Defined in outer scope
- Bound Variable: Bound to function argument in scope
- Closed: Term w/o. free variables
- Open: Term w. free variables
- Dynamic Scoping: Prevent capture during substit.
- Alpha Equivalent: Terms that differ only by consistent renaming of bound variables

Interpreter

- Done using operational semantics
- $exp \Downarrow v$: exp evaluates to v
- Short Circuit: Prevents storing values when are only needed in a short lived expression
- Typing:
  - Subtype Relation: Pos <: Int • Partial Order
  - Subtype Hierarchy: Tree of relations
  - Least Upper Bound (LUB): Lowest common parent of two types • LUB(T, F) = Bool
  - Alg. join: $T_1 \vee T_2$
  - Soundness of Subtyping Relation: $T_1 <: T_2$ is sound if it approximates the underlying semantic subset relation • write $[\![T]\!]$ for the subset of (closed) value of type T: $[\![T]\!] = \{v \mid \vdash v : T\}$
  - e.g. $[\![Zero]\!] = \{0\}$, $[\![Pos]\!] = \{1, 2, 3 \ldots\}$.
  - If $T_1 <: T_2$ implies $[\![T_1]\!] \subseteq [\![T_2]\!]$, then $T_1 <: T_2$ is sound
  - $[\![LUB(T_1, T_2)]\!] \supseteq [\![T_1]\!] \cup [\![T_2]\!]$
  - Subsumption: $\frac{E \vdash e : T \quad T <: S}{E \vdash e : S}$
    Treat val of type T as S whenever $T <: S$
  - Subtype Function: $\frac{S_1 <: T_1 \quad T_2 <: S_2}{(T_1 \to T_2) <: (S_1 \to S_2)}$
  - Depth Subtyping: $T_1 <: U_1 \quad T_2 <: U_2 \quad \ldots \quad T_n <: U_n$
    $\{lab_1 : T_1; lab_2 : T_2; \ldots; lab_n : T_n\} <: \{lab_1 : U_1; \ldots; lab_n : U_n\}$
  - Width Subtyping: $m \leq n$
    $\{lab_1 : T_1; lab_2 : T_2; \ldots; lab_n : T_n\} <: \{lab_1 : T_1; \ldots lab_m : T_m\}$
  - Mutable Structures: are invariant: $T_1 \neq T_2 \Rightarrow$ ref $T_1 \neq$ ref T_2
- Classes & objects: Objects contain a ptr to a dispatch vector (vtable)
  - Single Inheritance: Every method has ID (unique in class)
  - ID is used to lookup method in vtable
  - Multiple Inheritance: Multiple Solutions:
    - 1) Multiple Dispatch Vectors: Choose vtable based on static type • Casting from/to a class may require runtime operations
      ⊕ efficient ⊖ complicated to program
    - 2) Search + Inline Cache: maintain table which maps (method name to ptr of code • recursively walk up the hierarchy looking for the right method • alternatively use hash table
      ⊕ Simple ⊕ efficient ⊖ wasted space in DV ⊖ extra argument needed for resolution ⊖ slows dispatch for conflicts
    - 3) Sparse DV Tables or Bin Search Tree: ⊖ know whole class hierarchy ⊕ efficient





Optimisations

- Optimize • Time • Space • Power
- Done in different stages of the compile
- Inlining: Replace func call with func body
- Function Specialisation: Create specialized version of func that is called from diff. places w. diff. args
- Constant Folding: If ops are statically known, compute value at compile time • Performed at every stage of optimization • Ex: `int x = (2+3)*y` → `int x = 5*y`
- Constant Propagation: Replace usage of const var by value (use const. folding)

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

- Value Numbering:
- Dead Code Elimination (DCE): If side-effect free code can never be observed, eliminate it • Ex:

```
x = a*b;   // x is dead
  ⋮        // x never used →  ⋮
x = c+d;                     x = c+d;
```

- Unreachable Code Elimination: Remove unreachable basic blocks
- Loop-Invariant Code motion: Hoist invariant code out of loop. • Ex:

```
while (a) { b = c*d; ... // c,d not updated }
b = c*d; while (a) { ... // c,d not updated }
```

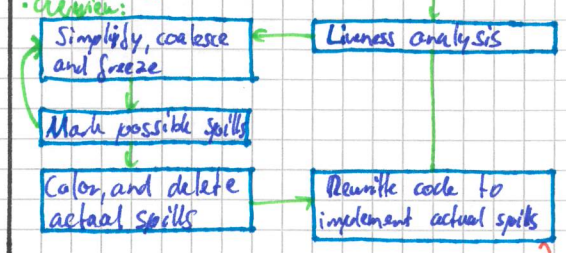
- Common Subexpression Elimination (CSE): Fold redundant computations together • Ex:
  `a[i] = a[i] + 1` compiles to `t = a+i*4; [t] = [t] + 1`
  `t = a+i*4; [t] = [t] + 1`
- Algebraic Simplification: Use mathematical simplification rules when are sound • Identities: Ex: $a+1 \to a$
- Reassociation & Commutativity: Ex: $(a+1)+2 \to a+(1+2) \to a+3$ • Strength Reduction: Replace expensive op with cheaper op • Ex: $a*4 \to a<<2$
- Careful with float/int arit. due to rounding / over-/underflow
- Strength Reduction (for loops): Repl. * by +
- Constant Folding & Propagation
- Branch Prediction/Optimisation
- Register Allocation: Assignment of regs
- Loop Unrolling:
- Cache Optimization
- Copy Propagation: If y is assigned to x, replace x's uses with y • Ex:

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

Code Analysis / Dataflow Ana.

- Liveness Analysis: Compute live vars between each statement
- Live: Is var v at program point L if:
  - v is defined before L • v is used after L
- For each node (=statement) is define:
  - $use(s)$: Set of variables used by s
  - $def(s)$: set of variables defined by s
- Live: Is var v at edge e if there is (both):
  - 1: a node n in the CFG s.t. $use(n)$ contains v
    - i.e. v will be used on some path starting from e
  - 2: a directed path from e to n s.t. for every statement s' on the path, $def(s')$ does not contain v
    - i.e. v will not be redefined on that path before the use
- For each node (=statement) is define:
  - $in(s)$: set of variables live on entry to n
  - $out(s)$: Set of variables live on exit from n
- Constraints: • $in(n) \supseteq use(n) \cup (out(n) \setminus def(n))$
  • $out(n) \supseteq in(n')$ if n' is successor of n
- Idea: Start with coarse guess and improve with iterations
- Algo:
  for all n: $in(n) := \{\}$, $out(n) := \{\}$
  repeat until no changes in 'in' and 'out':
  for all n:
  $out(n) := \bigcup_{n' \in succ(n)} in(n')$
  $in(n) := use(n) \cup (out(n) \setminus def(n))$
- Improve: If a node n's successors have not changed, n will not change ⇒ keep track of which node's successors have changed using FIFO queue

- Register Allocation: Find mapping from temps to regs s.t.: • program semantics are preserved • reg usage is maximized • minimize between reg moves • obey calling convention
  • Stack Spilling: spill excess temps to stack
- Interference graph: • Nodes: are temps • Edges: between nodes of interfering variables • Two nodes interfere if their live ranges intersect
- Kempe's Algo: Find K-coloring recursive in 3 steps:
  - 1: Find node with $deg < K$ and cut it out of the graph (remove all its edges) ⇒ simplified graph
  - 2: Recursively K-color the remaining graph
  - 3: When remaining graph is colored, color cut-out node using a free color (there is always such one)
  - If step 1 fails bc $deg \geq K$ $\forall n$, spill one temp:
    • Pick: node with highest (or high) interference
    • mark it and remove it from graph • continue algorithm • may still be able to color it at the end
- Access Spilled Regs: Two solutions:
  - 1: reserve two regs for moving from/to memory
    ⊖ loose two regs ⊕ need to color just once
  - 2: rewrite program to use a new temp with explicite move to/from memory ⊕ all regs available
    ⊖ add temporary changes to liveness ⇒ need to recompute liveness and recolor graph
- Precolored Nodes: certain regs must be used for certain cases ⇒ pre-assign color and treat node as having infinite degree ⇒ prevent cut-out of node
- Move Related Edges: for many T1 T2, if we assign T1 T2 the same reg, this instruction can be removed
  - Solution 1: add move related edge between such nodes • when color such node try to respect the color •
  - Solution 2: coalesce such nodes • is more aggressive as some coloring is forced ⊖ may prevent k-coloring: two algos which preserve k-coloring:
    - Briggs' Strategy: Safe to coalesce x & y if resulting node has fewer than k neighbours that have deg ≥ k
    - George's Strategy: Safe to coalesce x & y if $\forall$ neighbour t of x, either t already interferes with y, or t has deg $< k$
    - use: George if one of x & y is precolored
      Briggs if both are temporaries



- **Reaching Definition Analysis:** What var definitions reach a particular use of the variable • Used for:
  - constant propagation & copy propagation
- Define: • **defs[a]:** Set of nodes that define var a
  - gen(n) = {n} if quadruple n is an assignment, else = {}
  - kill[n] = defs[a] \ {n} if quadruple n is an assignment, else = {}
- Constraints: • out(n) ⊇ gen(n) • in(n) ⊇ out(n') if n' is predecessor of n • out(n) ∪ kill(n) ⊇ in(n)
- Initial: in(n) = out(n) = {} ∀ n
- Update: $in(n) := \bigcup_{n' \in pred(n)} out(n')$, out(n) := gen(n) ∪ (in(n) \ kill(n))
- **in(n):** Set of nodes defining some var such that the definition may reach the beginning of node n
- **out(n):** Set of nodes defining some var such that the definition may reach the end of node n

- **Available Expressions:** For common subexpression elimination
- Define: • **in(n):** set of nodes whose values are available on entry to n • **out(n):** set of nodes whose values are available on exit of n • **uses[a]:** set of nodes that use var a

	• gen(n)	• kill(n)
a = b op c	{n} \ kill(n)	uses(a)
a = load b	{n} \ kill(n)	uses(a)
store b, a	{}	uses(x) ∀ x that may be equal a
$a = f(b_1 \dots b_n)$	{}	uses(a) ∪ uses(x) ∀ x
$f(b_1 \dots b_n)$	{}	uses(x) ∀ x
else	{}	{}

- Constraints: • out(n) ⊇ gen(n) • in(n) ⊆ out(n') if n' is predecessor of n • out(n) ∪ kill(n) ⊇ in(n)
- Initial: in(n) = out(n) = set of all nodes, ∀ nodes
- Update: $in(n) := \bigcap_{n' \in pred(n)} out(n')$, out(n) := gen(n) ∪ (in(n) \ kill(n))

- **Overview:**

	May	Must
Forward	Reaching defs	Available expressions
Backward	Live vars	Very busy expressions

- (Forward) Data flow analysis Framework:
  - 1: Domain of data flow values $\mathcal{L}$
  - 2: For each node n, a flow function $F_n : \mathcal{L} \to \mathcal{L}$ • i.e. $F_n(\ell) = gen(n) \cup (\ell \setminus kill(n))$, $out(n) = F_n(in(n))$
  - 3: Combine operator ⊓ • i.e. $in(n) = \bigsqcap_{n' \in pred(n)} out(n')$
  - Initial: in(n) = out(n) = Top ∀ n • **Top:** has maximal information • $\ell_1 \sqsubseteq \ell_2$: ℓ_2 provides at least as much information as ℓ_1 • $\mathcal{L}$ is a poset under ⊑
- General: $\square(n) = \bigsqcap_{n' \in succ(n)} \square(n')$, $\square(n) = gen(n) \cup (\square(n) \setminus kill(n))$
- Forward: □ = in, □ = out, □ = pred • May: ⊓ = ∪
- Backward: □ = out, □ = in, □ = succ • Must: ⊓ = ∩

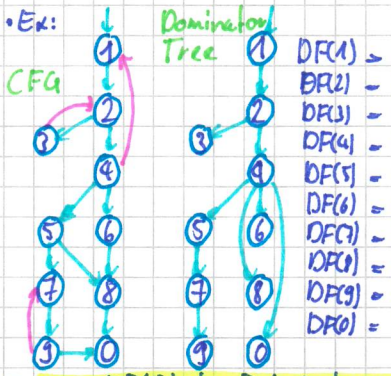
- **Best Possible Solution:** Suppose we have CFG
- Best solution at the output is some $\ell \sqsubseteq \ell_p$ ∀ paths p
- **Meet-over-paths (MOP) Solution:** $\bigsqcap_{p \in paths(n)} \ell_p$
- Iterative solution computes MOP solution iff flow functions F_c distribute over ⊓
  - Distribute: $\bigsqcap_i F_n(\ell_i) = F_n(\bigsqcap_i \ell_i)$
  - all 4 solutions compute MOP solution
  - Calculating MOP costs $O(2^n)$ for O(n) nodes and edges
  - Iterative costs $O(n^2)$ or so

Code Analysis / Control-Flow Ana.

loops with same header are merged

- **Loop:** Set of nodes in CFG • **Header:** only entry point • **Exit node:** ~~only~~ exit points
- **Strongly Connected Component (SCC):** is loop
- **Domination:** ~~Inverse~~ A dominates B iff only way to reach B from start node is via A
  - **Properties:** • Transitive • Antisymmetric
- **Back Edge:** Edge if target dominates the source
- **Dominator Tree:** Exists ∀ CFGs • **Node:** Node in CFG • **Edge:** if Source dominates destination directly (i.e. least dominator)
- **Dominator Data flow analysis:** • **dom(n):** Set of nodes that dominate n • $in(n) = \bigcap_{n' \in pred(n)} out(n')$ • out(n) = in(n) ∪ {n} • ⊤ = set of all nodes • $F_n(x) = x \cup \{n\}$ • ⊓ is ∩ • is MOP
- **Optimisation:** • dom(b) = nodes along start node to n path, in domination order • doms(n) = immediate dominator of of n • calculate dom(n) by walking doms(n)
- Dominator analysis identifies back edges
- **SSA:** • **Problem:** How to name variable x to which we assign a value in both branches of if/else?
- **Phi Function:** Fictitious operator used for analysis • Choose version of a variable by the path how control enters phi node
- %x4 = phi <ty> v_1, <label$_1$>, ..., v_n, <label$_n$>
- e.g. %x4 = phi i64 %x2, %then, %x3, %else takes %x2 or %x3 depending on branch and stores into %x4
- **Dominance Frontier:** of node B is the set of all CFG nodes y s.t. B dominates a predecessor of y, but does not strictly dominate y • A **strictly dominates** B if A dom B & A ≠ B
  - DF(n): dominance frontier of n
  - I.E. start at B, there is a path to y, but there is another route to y that does not go through B ⇒ y ∈ DF(B)

- Ex:



- y ∈ DF(B) if: • B dominates predec. of y • B does not strictly dom y
- Algorithm: for computing DF(n)
  for all nodes B
    if # pred(B) ≥ 2 // optimisation
      for each p ∈ pred(B)
        runner := p // start at predecessor of B
        while (runner ≠ doms(B)) // walk up the dominator tree
          DF(runner) := DF(runner) ∪ {B}
          runner := doms(runner)
- Insert ϕ at join points
- Suppose variable x is defined at a set of nodes N • $DF_0(N) = DF(N) = \bigcup_{n \in N} DF(n)$ • $DF_{i+1}(N) = DF(DF_i(N) \cup N)$
- Let J(N) be the least fixed point of sequence $DF_0(N) \subseteq DF_1(N) \subseteq \dots$ ⇒ insert ϕ function for var x at each node in J(N)

Automatic Memory Management

- **Unreachability:** Obj x is unreachable one of: • a reg contains a ptr to x • another reachable obj contains a ptr to x
- **Garbage:** unreachable object
- **Garbage Collection Scheme:** • 1: allocate space as needed • 2: when space runs out, remove garbage
- **Mark and Sweep:** When out of Mem, exec. 2 phases
- **Mark Phase:** Trace reachable objects • add all root pointers to todo list • Set mark bits of all object to 0 • go through todo list • Set mark bit to 1 when visiting obj. • Recursively add pointers to todo
- **Sweep Phase:** Collect garbage • add all objs with mark bit 0 to free list • reset all mark bits
- **Difficulty:** Need to store todo and free list in the objects itself, since we are out of Mem • **Ptr reversal:** when ptr followed, reverse it
- ⊕ no need to copy objs ⊖ fragments Mem

- **Stop and Copy:** Mem is split into two areas:
  - **Old Space:** for allocations • **New Space:** reserve for GC
- **Heap Ptr** points to next free word in old space
- **Working:** • old space is full • partition new space:

  - copy obj at root & set forward ptr to new obj
  - Scan copied obj for ptrs and recursively copy them
  - Once scan points to alloc, we are done and old space and new space switch roles
- ⊕ fast ⊕ allocation & collection are cheap
- ⊖ need to copy objs
  - ⊖ does not allow copy as given a mem loc. we cannot determine if it is a ptr ⇒ cannot update ptrs
  - **Conservative:** assume everything that looks like a ptr to be one ⇒ overestimate reachable obj
- **Reference Counting:** as soon as there is no ref to an object, it is garbage • do not wait till memory runs out • Implemented using ref counter in obj ⊕ easy to implement ⊕ works on-the-go ⊖ manipulate ref count at each assignment is slow ⊖ cannot collect circular structures
- **Concurrent:** allow program to run while collection happens
- **Generational:** do not scan long-lived objs at every collection
- **Parallel:** several collectors work in parallel